UNIVERSITY OF BERGEN

DEPARTMENT OF INFORMATICS

ALGORITHMS

Complexity of Edge-Editing to a Connected Graph of Bounded Degrees

Student:
Øyvind Stette HAARBERG

Supervisor:
Petr A. GOLOVACH

Master Thesis

2019

Abstract

In the EDGE EDITING TO A CONNECTED GRAPH OF BOUNDED DEGREES problem we are given a graph G, an integer k and a function f that assigns each vertex in G an integer bound on the degree. The task is to answer if there exists some connected graph H on the same set of vertices such that for every vertex in H the degree is within the function bound, and the size of the symmetric difference between the edge set of G and the edge set of H is at most k. In this thesis we have considered both an upper bound and a lower bound. For the upper bound we show that the problem is NP-complete, and give a polynomial kernel with $\mathcal{O}(k^3)$ vertices and $\mathcal{O}(k^4)$ edges. In addition we give a $2^{\mathcal{O}(k)} \cdot |V(G)|^{\mathcal{O}(1)}$ FPT algorithm based on random separation, which we show is asymptotically optimal assuming ETH. For the lower bound we give a polynomial algorithm using matching techniques.

Acknowledgements

First of all I would like to thank my supervisor, Petr Golovach, for all his guidance and help during the writing of this thesis. I would also like to thank my fellow algorithms students for all the help and fun distractions throughout my studies.

Finally I would like to thank my family for all the support thoughout the years.

Contents

Chapter 1

Introduction

Graphs can be used as a powerful tool for modeling several kinds of abstractions. There are several notions of graphs, but in general they consist of some *vertices* joined together by *edges*. Whenever one models something in a graph the vertices are some distinct elements, and the edges represent relationships between these elements. A lot of theoretical computer science is concerned with the study of graphs and their properties, which can give us useful information or knowledge about any other structure which can be modeled as a graph.

Sometimes we are interested in how *far* a graph is from some desired property. This question can be viewed as an *editing* problem, where we ask if the graph can be modified a limited number of times to achieve the desired property. The changes we consider is some set of allowed *editing operations*, where the more common ones are *vertex deletion*, *edge deletion* and *edge addition*. Other operations that might be considered include *vertex addition* and *edge contraction*. Some well-known problems of this variety are FEEDBACK VERTEX SET where we want to remove a certain number of vertices from a directed graph and obtain a graph without cycles, or CLUSTER EDITING in which we want to obtain a graph where every component is a clique. Other problems that might not seem to be editing problems on the surface have natural formulations as editing problems: In the MINIMUM VERTEX COVER problem where we given a graph, and want to find a minimum set of vertices (a vertex cover) such that every edge in the graph is incident to at least one of the vertices of the vertex cover, we can instead ask for a minimum set of vertex deletions such that the graph after edits is an independent set. For HAMILTONIAN CYCLE where the question is whether there exists a cycle on all the vertices in the given graph, we can instead ask whether there exists some set of edge deletions such that the resulting graph is a cycle on all vertices in the graph.

We can divide the properties we are looking for in graphs between *hereditary* and *non-hereditary non-trivial* properties. A hereditary property is a property that if it holds for some graph G, then it also holds for any subgraph $H \subseteq G$. A non-hereditary property is a property where this does not hold. A non-trivial property is a property that holds for an infinite number of graphs, and does not hold for an infinite number of graphs. An example of a hereditary graph property is having a vertex cover of size k: If we can cover all edges with k vertices in G then it must also be possible for any subgraph of G. An example of a non-hereditary property is *connectivity*, as even though G is a connected graph a subgraph might exclude some critical set of vertices or edges such that the subgraph is not connected.

An early landmark in the study of graph editing problems was by Lewis and Yannakis [17], when they showed that graph editing for non-trivial hereditary properties by vertex deletion is NP-complete. When considering editing by edge-addition or -deletion to reach some graph class, Natanzon, Shamir and Sharan [22] proved the NP-hardness for several graph classes, which was later extended by Burzyn, Bonomo and Durán [1].

As graph editing problems are often NP-complete the framework of parameterized complexity as pioneered by Downey and Fellows [8] is useful for further analyzing the complexity of editing problems. Cai [2] showed that editing problems with vertex deletion, edge addition and edge deletion are fixed parameter tractable for hereditary properties if the properties can be be characterized by a finite number forbidden induced subgraphs. The results for hereditary properties were later extended by Khot and Raman [16] where they parameterized by the number of edits.

There are no broad results for editing problems when considering non-hereditary properties such as degree constraints or connectivity, but Moser and Thilikos [21] studied the probem of whether it is possible to obtain a r-regular subgraph of some graph by deleting at most k vertices. This study was extended by Mathieson and Szeider [19], who studied several variations of DEGREE CONSTRAINT EDITING problems. In DEGREE CONSTRAINT EDITING you are given some graph G, a function $\delta : V(G) \to 2^{\{0,\dots,d\}}$ of permitted degrees, and a budget k for edits, and you want to answer if it is possible to edit the graph using at most k operations such that for all vertices v in the graph $d(v) \in \delta(v)$ holds. The editing operations they considered are some combination of vertex deletion, edge addition and edge deletion. As the function for permitted degrees is very general in that every vertex has a separate set of allowed degrees $\leq d$, they also looked at variations where the target degrees for every vertex was a singleton. In particular they showed that if the target degree for each vertex is a single integer and the permitted editing operations are edge additions and edge deletions the problem is solvable in polynomial time. Without any restriction on the set of permitted degrees they showed that the problem is W[1]-hard for any combination of the editing operations of vertex deletion, edge addition and edge deletion. Vertex deletions cause the problem to be W[1]-hard even for singleton degrees when parameterized by the number of edits only. When parameterizing by d and k they showed that the problem is FPT but did not provide an algorithm (the proof was non-constructive), and they also proved a polynomial kernel if only vertex or edge deletions are permitted in the case of singleton degrees.

Golovach [12] later gave an explicit FPT algorithm for DEGREE CONSTRAINT EDITING when parameterized by d and k for the editing operations of vertex deletion, edge deletion and edge addition, but showed that unless NP $\subseteq$ co-NP / poly the problem does not admit a polynomial kernel with the same parameterization. As a natural extension to the classification of degree-constrained editing Golovach [13] looked at the problem of editing to a *connected* graph of given singleton degrees when the editing operations are edge deletion and edge addition (EDGE EDITING TO A CONNECTED GRAPH OF GIVEN DEGREES), and gave a polynomial kernel parameterized by $d + k$. The case when the target degree was the same for all vertices (EDITING TO A CONNECTED F-DEGREE GRAPH) was shown to be FPT parameterized by k. This result was later improved upon by Fomin et. al. [9] who gave a $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$ algorithm for this problem using color coding and matroids.

An interesting extension of EDGE EDITING TO A CONNECTED GRAPH OF GIVEN DEGREES is to consider the degree constraint as a bound and not a singleton value. This degree constraint is more free than giving an exact degree for every vertex, but at the same time more strict than having the very broad degree constraint considered by Mathieson and Szeider [19] for DEGREE CONSTRAINT EDITING. For instance one could have some data network with some existing connections between nodes, and some upper limitation for each node on how many different connections it can handle. The question of whether a connected network obeying the connectivity constraints of the nodes can be obtained by adding or removing at most k connections is the first of the two problems treated in this thesis, namely EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES, or EDITUBD for short.

EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES
Input: A graph $G = (V, E)$, an integer k, a function $f : V(G) \to \mathbb{Z}^+$
Question: Does a set $A \subseteq \binom{V(G)}{2}$ exist such that $G' = (V(G), E(G) \triangle A)$ is a connected graph, all vertices v in $V(G')$ has $d(v) \leq f(v)$, and $|A| \leq k$?

In this thesis we give a polynomial kernel for this problem with $\mathcal{O}(k^3)$ vertices and

$\mathcal{O}(k^4)$ edges. As the output of the kernelization algorithm has a lot of structure we are also able to give a $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$ algorithm based random separation techniques as pioneered by Cai et. al. [3]. The section on EDITUBD is rounded off with a proof of NP-completeness and giving a lower bound for the running time of any algorithm based on the *Exponential Time Hypothesis* (ETH), allowing us to conclude that a single exponential time algorithm is asymptotically optimal assuming ETH.

The other problem considered in this thesis is the lower bounded version of EDITUBD: EDGE EDITING TO A CONNECTED GRAPH OF LOWER BOUNDED DEGREES, or EDITLBD for short. This problem can for instance map to a problem where you are given some communication network with existing links between entities, and each entity in the network has some minimum requirement of the number of other entities it can contact directly. The problem of deciding if at most k connections can be added or removed to fulfill the individual requirements of the entities and also connect the entire network together can be formulated as an instance of EDITLBD.

EDGE EDITING TO A CONNECTED GRAPH OF LOWER BOUNDED DEGREES

Input:	A graph $G = (V, E)$, an integer k, a function $f : V(G) \rightarrow \mathbb{Z}^+$
Question:	Does a set $A \subseteq \binom{V(G)}{2}$ exist such that $G' = (V(G), E(G) \triangle A)$ is a connected graph, all vertices v in $V(G')$ has $d(v) \geq f(v)$, and $\lvert A \rvert \leq k$?

In the chapter after we EDITUBD we provide a polynomial algorithm for EDITLBD based on matching techniques.

Chapter 2

Preliminaries

2.1 Set theory

A *set* is an unordered collection of distinct objects. Throughout this thesis we will use the common notation for sets with relations membership ($\in, \notin$), subset ($\subset, \subseteq$), union ($\cup$) and intersection ($\cap$). For two sets A and B the *symmetric difference* $A \triangle B$ is the union of A and B without the intersection of A and B ($A \triangle B = (A \cup B) \setminus (A \cap B)$).

Set builder notation will generally be used when defining sets in this thesis, but we will also refer to some common symbols for some useful sets: $\emptyset$ denotes the *empty* set, the set with no elements. $\mathbb{Z}$ refers to the set of all integers, with $\mathbb{Z}^+ = \{0, 1, 2, \dots\}$ being the set of non-negative integers. $\mathbb{N}$ refers to the set of natural numbers $\{1, 2, 3, \dots\}$. Another way we construct sets is by choosing all subsets of a certain size from another set. If A is a set and k is the size of the subsets of A we want, $\binom{A}{k}$ is the set of all k-sized subsets of A.

We say that a set is *inclusion-wise maximal* or simply *maximal* according to some property if no strict supersets exist such that the superset also has the same property. Similarily we say that a set is *inclusion-wise minimal* or simply *minimal* if no strict subset exists according to some property. The biggest maximal set of all maximal sets that has some property is a *maximum* set, and similarily the smallest minimal set of all minimal sets that has some property is a *minimum* set.

2.2 Graphs

A *graph* G is a tuple $G = (V, E)$ of the set of vertices V and the set of edges E obeying $E \subseteq \binom{V}{2}$, such that the elements of E are sets with two elements of V.

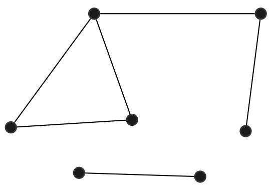

Figure 2.1: Example simple graph

Figure 2.1 shows a drawing of a simple graph, where the vertices are the circles, and the edges are drawn as lines between the vertices that they connect. The vertex set of a graph G is referred to as $V(G)$, and likewise the edge set of a graph G is referred to as $E(G)$. When there is no ambiguity we will use V and E to refer to the vertex set and edge set respectively. If the graph we are referring to is unambiguous we let n denote the number of vertices $|V|$, and m denote the number of edges $|E|$.

We say that an edge e is *incident* to a vertex v if $v \in e$. If an edge $e = \{u, v\}$ is in $E(G)$, we say that the vertices u and v are *adjacent*. Often we will refer to an edge by the vertices that it is incident to, so a uv-edge is an edge $\{u, v\}$.

In an undirected graph the sets are unordered sets, so the edge $\{u, v\}$ and $\{v, u\}$ is the same edge. This thesis deals only with undirected graphs. In a *simple graph* we do not allow self-loops, which are edges from and to the same vertex ($\{v, v\}$ is a self-loop on the vertex v). In this thesis we only consider finite graphs, so for any graph $|V|$ and $|E|$ is finite.

A *multigraph* is a graph $G = (V, E)$ where we can have multiple edges between the same pair of vertices. We do this by introducing a mapping function $E \to \binom{V}{2} \cup V$ that assigns each edge the vertices it is incident to. This allows us multiple "copies" of the same edge, as two edges can be between the same pair of vertices. For convenience we will assume that the mapping function is defined for the edges we are working with in a multigraph, such that we will never explicitly refer to this mapping function but instead refer to the edges by the vertices they map to – even if referring to the vertices no longer uniquely identifies an edge. In multigraphs we allow for (multi-)self-loops, which are edges of the form $\{v, v\}$. The *multiplicity* of a $\{u, v\}$-edge in a multigraph is the number of edges in the edge set that maps to the vertices u and v. Figure 2.2 shows a drawing of a multigraph with some multiedges and self-loops.

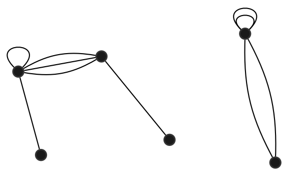

Figure 2.2: An example multigraph with self-loops

If G is a graph and S is a subset of $\binom{V(G)}{2}$ (or a set of edges that map to sets in $\binom{V(G)}{2} \cup V(G)$ in the case of multigraphs), then $G - S = (V(G), E(G) \setminus S)$, and $G + S = (V(G), E(G) \cup S)$. If G is a graph and T is a set of vertices, then $G - T = G'$ with $V(G') = V(G) \setminus T$ and $E(G') = \{\{u, v\} \in E(G) \mid u, v \notin T\}$. For convenience we will sometimes be more imprecise and write $G - e$ or $G + e$ instead of $G - \{e\}$ or $G + \{e\}$ if e is a single element.

If G is a graph and A is a subset of $\binom{V(G)}{2}$ (or "edges" that map to vertices in $\binom{V(G)}{2} \cup V(G)$ in the case of multigraphs), we define the *symmetric difference* $G \triangle A$ as the graph $G' = (V(G), E(G) \triangle A)$, where every edge e that is in both $E(G)$ and A is removed, and every edge that is in A but not in $E(G)$ is added. Informally we refer to the edges of A that are in G as *edge deletions*, and edges of A that aren't in G as *edge additions*.

For the definitions of $G - S$ and $G \triangle A$ we abuse notation somewhat, and refer to the elements of A or S as edges even though some elements in A or S aren't edges of G.

We define the *degree* $d_G(v)$ of a vertex v in a graph G to be the number of edges in $E(G)$ where v is one of the vertices. If the graph we are referring to is unambiguous we skip the G in the subscript. For a self-loop $\{v, v\}$ we count v twice as it has two endpoints, so each self-loop contributes 2 to the degree of a vertex.

For a simple graph G we define the *complement* $\overline{G}$ of a graph G as the graph on the same vertices, but for every pair of vertices connected by an edge in G there is no edge in $\overline{G}$, and for every pair of vertices not connected by an edge in G they are connected by an edge in $\overline{G}$.

A *subgraph* $G' \subseteq G$ is a graph where $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. An *induced subgraph* $G[S]$ of G (where S is a set of vertices) is a subgraph of G where the vertex set is the intersection of $V(G)$ and S, and the edge set is the set of edges where both endpoints are in S. An *edge-induced subgraph* $G[T]$ of G on a set of edges $T \subseteq E(G)$ is a subgraph G' with $E(G') = T$ and $V(G') = \bigcup_{\{u,v\} \in T} \{u, v\}$. We use the same notation for subgraphs induced on both a set of vertices and a set of edges.

A *path* is an ordered sequence of distinct vertices where for every adjacent pair of vertices in the sequence there exists an edge betwee these two vertices. A u, v-path is a path starting at vertex u and ending in vertex v. A *cycle* is an ordered sequence of vertices where the first and last vertex are the same, all the other vertices are distinct, and for every pair of adjacent vertices in the sequence an edge between these vertices exist in the graph. If we have self-loops in a graph a self-loop is a cycle.

Two vertices u, v are *connected* if there exits a path from u to v. We also consider a vertex to be connected to itself, even without self-loops. A *connected component* $C = G[A]$ where $A \subseteq V(G)$ is an induced subgraph on a maximal set of vertices A such that for every two vertices in C there exists a path between them in C. A connected component can be a single vertex. We say that a graph as a whole is *connected* if there is only one connected component in the graph. We use the shorthand $\mathcal{C} = \bigcup\limits_{C_i \in G} \{C_i\}$ where each C_i is a connected component to denote the set of connected components of a graph G. Two vertices u, v are edge-biconnected if after the removal of any edge in G, u and v are still connected.

An *independent set* is a set of vertices I such that the induced subgraph $G[I]$ has no two distinct vertices that are pairwise adjacent. Note that with this definition we might have edges in a graph induced on an independent set in the form of self-loops.

If for an edge e there exists two vertices u, v such that u and v are connected in G, but in $G - e$ the they are not connected, the we call e a *bridge*, or a *cut edge*. If an edge is not a bridge, then it must be a part of some cycle, and we call e a *non-bridge edge* or a *cycle edge*.

Trees are connected graphs without any cycles. A tree with n vertices has exactly $n - 1$ edges, and every edge is a bridge [6, 1.5.3]. If all the connected components of a graph are trees then the graph is a *forest* (thus a tree is also a forest with only one component). A *spanning graph* H is a subgraph of some graph G with all the vertices of G ($V(H) = V(H)$), but not necessarily all the edges. A *spanning forest* of a graph G is a spanning subgraph of G which is a forest with the same number of connected components as G.

We can *contract* an edge e between two distinct vertices u, v by replacing u and v with a new vertex v'. The edge e is removed from the graph, and all other edges originally incident to u or v are now incident to v' instead. If there are multiedges parallel to e these become self-loops on v'. Figure 2.3 shows an example contraction of an uv-edge.

We define merging of two non-adjacent vertices u, v as follows: Let v' be the new vertex representing the merging of u and v, so $V(G') = (V(G) \setminus \{u, v\}) \cup \{v'\}$. Every non-loop edge $\{u, c\} \in E(G)$ or $\{v, c\} \in E(G)$ is replaced the edge $\{v', c\}$ to $E(G')$. Every loop $\{u, u\}$ or $\{v, v\}$ is replaced by the loop $\{v', v'\}$.

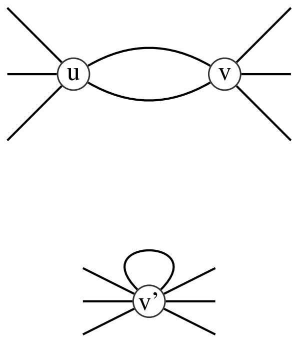

Figure 2.3: Example contraction

2.3 Algorithms and Complexity theory

We will now introduce some definitions that allow us to formally define a parameterized decision problem. An *alphabet* is a fixed and finite non-empty set of symbols, commonly denoted as Σ. All possible strings on an alphabet is denoted as Σ^*. For any given string I we say that $|I|$ is the *length* of I, measured as the number of symbols in the string. A *language* L over an alphabet Σ is a subset of all the strings in our alphabet, $L \subseteq \Sigma^*$.

We define a *decision problem* as a language L over some alphabet Σ^*. For any string $I \in \Sigma^*$, if $I \in L$ we say that I is a YES-instance, and if $I \notin L$ we say that I is a NO-instance of the language L. A *parameterized decision problem* is some language $L \subseteq \Sigma^* \times \mathbb{N}$, where for an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is the *parameter*. Similarily as for decision problems we define that for an instance $(x, k) \in \Sigma^* \times \mathbb{N}$ of a parameterized problem, (x, k) is a YES-instance if $(x, k) \in L$, and a NO-instance if $(x, k) \notin L$. We will informally refer to both parameterized decision problems and decision problems as simply *problems*.

In this thesis we will present *algorithms* for decision problems. An algorithm can be formalized in the framework of *Turing machines*, which is a mathematical model of computation. We will be slightly informal and regard an algorithm as some computational procedure to solve some problem. For a more comprehensive introduction to Turing machines, languages and decision problems we refer to the book *Introduction to the Theory of Computation* by Sipser [23]. As we are dealing with decision problems, our algorithms for those will take as input some instance I, the *problem instance* or simply *instance*, and answer either YES or NO upon halting. The problems we deal with in this thesis deal with graphs, but we can encode any object including graphs and functions as strings.

2.3.1 Running time

To understand the complexity of an algorithm we will look at the *running time* of the algorithm. We measure the running time as the number of basic computational steps the algorithm takes to give an answer to an instance. The number of steps taken can vary depending on the input, so the running time of an algorithm is measured as some function $f : \mathbb{N} \to \mathbb{N}$ of the size of the input. Giving an upper bound on the runtime of an algorithm is captured by big-O notation: If we have an algorithm A with running time $f(|I|)$, and a function $g : \mathbb{N} \to \mathbb{N}$, if $f(|I|) \in \mathcal{O}(g(|I|))$ we say that the running time of A is bounded by $g(|I|)$.

We group the running time complexity of problems in complexity classes, where for our thesis the especially relevant ones are P, the problems that can be decided in time bounded by some polynomial in the size of the input ($f(x) \in |x|^{\mathcal{O}(1)}$), NP, the problems that can be decided in time bounded by some polynomial on a non-deterministic Turing machine, and FPT, the parameterized problems that can be decided in time $f(x, k) \in g(k) \cdot |x|^{\mathcal{O}(1)}$ for some computable function g. NP can equivalently be defined as the class of languages that can be *verified* in polynomial time [23].

A motivation for the complexity class of FPT, or Fixed Parameter Tractable is to gain a better understanding of the complexity of problems that are in NP. For instance, the NP-complete problem of finding a set of vertices S of size k in a graph G such that $G - S$ is an independent set (the VERTEX COVER problem [11]) can be done in time $2^c k \cdot n^{\mathcal{O}(1)}$ for some constant $c > 0$, meaning that vertex cover is FPT [4]. However the problem of finding a coloring of the vertices of a graph using at most k different colors such that no edge is incident to vertices of the same color (the VERTEX COLORING problem [11]) for $k \geq 3$ has no algorithm of the form $f(k) \cdot n^{\mathcal{O}(1)}$ unless P = NP, as 3-COLORING is NP-complete [11].

This apparent difference in complexity for NP-complete problems are captured in the *W-hierarchy* [7], which defines a hierarchy of complexity classes within NP, with

$$\text{W[0]} \subseteq \text{W[1]} \subseteq \text{W[2]} \subseteq \cdots \subseteq \text{XP} \subseteq \text{NP}$$

A working hypothesis is that W[i] $\subset$ W[j] for all $i < j$. W[0] is the complexity class of FPT, and W[1] is the lowest class of problems in the hierarchy that are believed to not be FPT. An example of a problem that is complete for W[1] is CLIQUE [5, Theorem 13.18], where the problem is finding a set of vertices S of size k such that every two vertices in $G[S]$ is pairwise adjacent. Showing that a problem is at least as hard as any problem in W[1] would mean that any FPT algorithm for the problem causes some collapse in the W-hierarchy. XP is the class of parameterized problems solvable in time $\mathcal{O}(|x|^{f(k)})$ for an instance (x, k) and some computable function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$.

For a more thorough treatment of parameterized complexity we refer to the book *Parameterized Algorithms* by Cygan et. al. [5].

2.3.2 Reductions

To better understand the complexity of problems it is useful to compare the difficulty of different problems with one another. One way of doing this is by introducing *reductions* from some problem A to another problem B. A *safe reduction* is a function $\phi : \Sigma^* \to \Sigma^*$ such that for any instance I, I is a YES-instance of A if and only if $\phi(I)$ is a YES-instance of problem B. If the running time of ϕ can be bounded by some polynomial in the size of the instance, we say that ϕ is a *polynomial reduction*. We can construct algorithms as a series of *reduction rules*, which are reductions to and from the same problem. We say that a reduction rule is safe if the reduction rule preserves the instance, i.e. $I \in L$ if and only if $I' \in L$.

The existence of some polynomial reduction from problem A to problem B means that problem A can be solved by any algorithm that solves B, in addition to the running time of the reduction function. This is useful for showing that problems belong to the class *NP-hard*, which is the class of the problems at least as hard as any problem in NP. The problems that are both in NP and are NP-hard belong to the class NP-complete, the hardest problems in NP. If we have a problem A that is known to be NP-complete, a polynomial time reduction from A to B means that we can solve A by first reducing to B in polynomial time, and then solving B, and therefore B must be NP-hard.

Another type of reductions we consider in this thesis are *kernelization algorithms* (or simply *kernels*), which is a particular kind of *parameterized preprocessing* algorithms. A parameterized preprocessing algorithm takes some instance (x, k) of some parameterized problem A and outputs an instance (x', k') such that (x, k) is a YES-instance of A if and only if (x', k') is a YES-instance of A. If a preprocessing algorithm ϕ runs in polynomial time, and we can bound the maximum size of the output instance (x', k') for any input instance by some computable function $g : \mathbb{Z}^+ \to \mathbb{Z}^+$ depending only on k, we say that ϕ is a kernelization algorithm. Here we have no requirement for the function g other than computability, but in practice we look for kernels bounded by a polynomial function g, which we call *polynomial kernels*. Kernels fit into the notion of FPT, since a problem has a kernel if and only if it also has a FPT algorithm [5, Lemma 2.2].

For a more in-depth look at kernels we refer to the books *Parameterized Algorithms* by Cygan et. al. [5], and *Kernelization: Theory of Parameterized Preprocessing* by Fomin et. al. [10].

To show that problems belong in the same class in the W-hierarchy polynomial reductions are no longer sufficient. This is because any NP-complete problem can be reduced to any other NP-complete problem using a polynomial reduction, and we have no way of separating the complexity classes. For this reason we also have *parameterized reductions* [5, Chapter 13.1]. A parameterized reduction takes some instance (I, k) of a parameterized problem A, and outputs an instance (I', k') of a parameterized problem B such that

1. (I, k) is a YES-instance of A if and only if (I', k') is a YES-instance of B
2. $k' \leq g(k)$ for some computable function g
3. the running time of the reduction is bounded by $f(k) \cdot |I|^{\mathcal{O}(1)}$ for some computable function f

Parameterized reductions are also useful for showing complexity lower bounds based on ETH [15].

Conjecture 2.3.1 (Exponential-Time Hypothesis, ETH)**.** *Let δ_q be the infinimum of the constants c for which there exists an algorithm solving q-SAT for $q \geq 3$ in time $\mathcal{O}^*(2^{cn})$. Then δ_3 > 0.*

If ETH holds we have a lower bound on the running time for any algorithm solving 3-SAT, as the consequence of ETH is that no subexponential algorithm can exist that solves 3-SAT. We can combine this with parameterized reductions from 3-SAT to other problems, and thus obtain lower bounds based on ETH.

2.4 f-matching

A f-matching is a generalization of matching, where instead of requiring a matching to at most have one edge incident to each vertex, we can have a vertex v matched $f(v)$ times. As for the general matching problem we can solve the problem of finding a maximum f-matching in polynomial time. Figure 2.4 shows an example graph where we want to find a maximum f-matching, where the function values f are indicated next to the vertices.

f-MATCHING

Input: A multigraph $G = (V, E)$ where the maximum multiplicity of any multi-edge is c and a function $f : V(G) \to \mathbb{Z}^+$

Output: A maximum set of edges $S \subseteq E$ s.t. every vertex v in $G[S]$ has degree $\leq f(v)$.

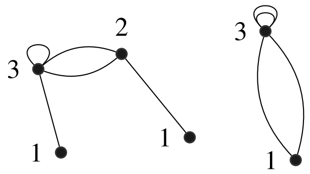

Figure 2.4: An example multigraph we want to find f-matching in

The problem and the construction used to solve it are from Lovasz' and Plummer's book on Matching Theory [18], but they consider the slightly different subproblem of finding a perfect f-matching (where every vertex v in G is matched exactly $f(v)$ times). We are interested in the maximum possible f-matching in G, but we will use the same construction as the one used to find a perfect f-matching to do this, adopted for multigraphs.

Constructing the matching graph: The graph G' which we will find a matching in is constructed as follows: For every vertex v in G, add $f(v)$ vertices $v_1, v_2, ..., v_{f(v)}$ to G'. For all non-self-loops e between vertices $\{u, v\}$ in G add two new vertices e_1 and e_2, and the edge $\{e_1, e_2\}$ to G'. Then add all edges between u_i and e_1, and all the edges between e_2 and v_j in G' for all is and js. For self-loops on a vertex u we do the same, except in this case both e_1 and e_2 are adjacent to all vertices u_i. We consider each multiedge as a separate edge. Figure 2.5 shows the graph G' corresponding to the graph in figure 2.4, where the duplicated $f(v)$ vertices are colored in blue, and the vertices added for the edges in G are colored in red. Note that G' will always be a simple graph, even though the input graph could be a multigraph.

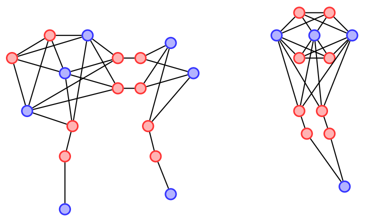

Figure 2.5: The graph G' corresponding to the example graph in figure 2.4

We have a mapping from a f-matching M in G to a matching M' in G' as follows: Every uv-edge e in M has its corresponding pair of $\{u_i, e_1\}$ and $\{e_2, v_j\}$-edges in M' for some yet unmatched u_i and v_j. For every uv-edge e in G that is not in M, either its corresponding $\{e_1, e_2\}$ edge is in M', or one of $\{u_i, e_1\}$ or $\{e_2, v_j\}$ is in M'.

Any maximal matching in M' that only matches one of the vertices e_1 or e_2 for some pair of vertices corresponding to an edge e in G can be altered to a maximal matching in M' of the same size where both these vertices are matched. This is done by removing the edge incident to the the matched vertex e_1 or e_2, and adding the edge $\{e_1, e_2\}$. For this reason we will assume that when we map from a maximal matching in G' for every $e \in G$ the corresponding vertices e_1 and e_2 are matched, and when we map from any f-matching M in G the vertices e_1 and e_2 are always matched in M'.

Lemma 2.4.1 (f-matching to matching). *We have a maximum f-matching M in G if and only if the corresponding matching M' mapped from M is a maximum matching in G'.*

Proof. We begin by showing that we can map any maximal f-matching M in G of size k to a matching M' in G' of size $k + |V(G)|$. Every uv-edge $e \in M$ causes us to have the edges $\{u_i, e_1\}$ and $\{e_2, v_j\}$ in M' for some i and j. Every uv-edge $e \in (E(G) - M)$ causes us to have the edge $\{e_1, e_2\}$ in M'. This mapping never matches any vertex created for any edges in G twice due to the fact that vertices e_1 and e_2 are only matched as part of the mapping from this particular edge, and the number of times we maximally match a vertex u_i due to this rule is $f(u)$, so every matched u_i can be distinct for any such vertex. From the mapping rule we know that for each edge $e \in E(G)$ we add either one edge to M' if $e \notin M$, or two edges if $e \in M$. Thus the size of the obtained matching M' must be equal to

$$|M'| = 2|M| + |E(G) - M| = |E(G)| + |M|$$

and it follows that if $|M| = k$ the size of M' must be $k + |E(G)|$.

We will now show that if we have a maximal matching M' of size $k + |E(G)|$ in G' the mapped f-matching M in G is a valid f-matching of size k. First, the mapped f-matching M from M' must be valid, as we only add the uv-edge e to M if we have edges $\{u_i, e_1\}$ and $\{v_j, e_2\}$ in M'. This can happen at most once for each edge, and the maximum number of vertices u_i that can be matched is exactly $f(u)$, so the mapped f-matching must be valid.

As we have a maximal matching in G', we can assume that all vertices e_1 and e_2 for every edge e in G are matched. If for an edge e in G the edge $\{e_1, e_2\}$ is in M', we do not add any edges to M. If for an edge e in G the we have the edges $\{u_i, e_1\}$ and $\{v_j, e_2\}$ in M', we add the edge e to M. Then, for every pair of e_1, e_2-vertices we either have two edges matched in M', or one edge matched in M'. As a consequence the minimum size of a maximal matching in G' is equal to $|E(G)|$. If we have two edges matched for a pair e_1, e_2 we add the corresponding edge e to the f-matching M. We have a total of $|E(G)|$ e_1, e_2-pairs in G', so the size of the mapped matching M from M' is equal to

$|M'| - |E(G)|$, and we have proven that a maximal matching of size $k + |E(G)|$ in M' maps to a f-matching in G of size k.

Finally we have to show that we have a maximum f-matching M in G of size k if and only if we have a maximum matching M' in G' of size $|E(G)| + k$. Assume that we have a maximum f-matching in G, but the mapped matching M' in G' is not a maximum matching. That means that we have a matching B in G' such that $|B| > |M'| = |M| + |E(G)|$. As B is a maximum matching it must also be maximal, so we can obtain a mapped f-matching B' in G of size $|B| - |E(G)| > |M| + |E(G)|$, and this is a bigger matching than the maximum matching we started with, which is a contradiction. The same argument works when initially starting with a maximum matching in G', so we have a maximum f-matching M in G if and only if the mapped matching M' is a maximum matching in G'.

□

The running time of this algorithm depends on the number of additional edges and vertices needed, as well as the running time of the matching algorithm used.

To give a reasonable bound on the running time of the algorithm we have introduced a parameter for the maximum multiplicity of multiedges c. We will therefore bound the running time by the number of vertices $|V|$, and the maximum multiplicity of any multiedge c. This analysis also works for simple graphs, as in a simple graph $c = 1$. We can also bound the function for any individual vertex v by setting $f'(v) = \mathit{min}(f(v), d(v))$ as our function instead, as we can never match a vertex more times than the number edges incident to it.

The number of additional vertices added is $\sum\limits_{v \in V(G)} f(v) \leq \sum\limits_{v \in V(G)} d(v) \leq |V|^2 \cdot c$, in addition to two new vertices per edge in the graph. For every vertex we add $d(v) \cdot f(v) \leq d(v)^2 \leq |V|^2 \cdot c$ edges, and we add one extra edge per edge in G.

Thus the maximum number of vertices in the matching graph is $2|E| + |V|^2 \cdot c$, and the maximum number of edges is $|V|^3 \cdot c^2 + |E|$. We know that $|E| \leq |V|^2 \cdot c$, which gives us a bound of $2 \cdot |V|^2 \cdot c + |V|^2 \cdot c \in \mathcal{O}(|V|^2 \cdot c)$ for the number of vertices in our matching graph, and $|V|^3 \cdot c^2 + |V|^2 \cdot c \in \mathcal{O}(|V|^3 \cdot c^2)$ for the number of edges in our matching graph.

The fastest general algorithm for finding the maximum matching in a graph is by Micali and Vazirani [20], achieving a running time of $\mathcal{O}(\sqrt{n} \cdot m)$ for a graph with n vertices and m edges. If we substitute in the bounds for the number of edges and vertices in our auxilliary matching graph we obtain a running time bound of

$$\mathcal{O}(\sqrt{|V|^2 \cdot c} \cdot (|V|^3 \cdot c^2)) = \mathcal{O}(|V|^4 \cdot c^{2.5})$$

We summarize these results in the following Theorem:

Theorem 2.4.2 (f-matching)**.** *Given a multigraph G with maximum edge multiplicity c and a function $f : V(G) \to \mathbb{Z}^+$, we can find a maximum f-matching M in G in time $\mathcal{O}(|V(G)|^4 \cdot c^{2.5})$. If G is a simple graph the maximum edge multiplicity c is 1, and the running time is then bounded by $\mathcal{O}(|V(G)|^4)$.*

Chapter 3

Upper Bounded Degrees

In this chapter, we will look at the EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES (EDITUBD) problem and its complexity. The problem was already stated for simple graphs in the introduction, but in this chapter we will work with the multigraph version.

3.1 Problem statement

EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES

Input:	A multigraph $G = (V, E)$, an integer k, a function $f : V(G) \to \mathbb{Z}^+$
Question:	Does a set A of edges exist, such that $G' = G \triangle A$ is a connected graph, all vertices v in $V(G')$ has $d(v) \leq f(v)$, and $\lvert A\rvert \leq k$?

When constructing algorithms it turned out to be useful to consider the multigraph version of EDITUBD. This is not an issue, as the answer for an instance where G is simple graph stays the same:

Lemma 3.1.1 (Preservation of instance)**.** *In the case where G is a simple graph, (G, f, k) is a* YES*-instance of* EDITUBD *when G' is restricted to simple graphs if and only if (G, f, k) is a* YES*-instance of* EDITUBD *where G' is permitted to be a multigraph.*

Proof. The forward direction of this equivalence follows from the fact that any simple graph is also a valid multigraph. If a simple graph G' can be constructed using $\leq k$ edits on G, the same set of edge deletions and additions is also a solution if multigraphs are permitted.

The reverse direction does not immediatly follow, as even though G initially is a simple graph, the solution graph G' could be a multigraph. If (G, f, k) is a YES-instance when multigraphs are permitted we know that we obtain a graph G' with at most k edits such that G' is connected and has no unsatisfied vertices. Assume that G' has some multiedge or self-loop e. As G initially has no multiedges or self-loops, the multiedge or self-loop must be from an edge addition. If G' is a connected graph with no violated degree constraints, then $G' - e$ must also be a connected graph with no violated degree constraints: The removal of e does not disconnect any components, and the degree constraint is an upper bound and thus the removal of any edge cannot cause a satisfied constraint to become violated. As e comes from an editing operation, omitting this operation will not make us go over budget. Then any solution that adds multiedges or self-loops will also be solutions if these multiedges or self-loops are not added.

□

3.1.1 Definitions

We will now define some notation used in this chapter in particular. For these definitions we let (G, f, k) be an instance of EDITUBD.

Definition 3.1.1 (Unsatisfied vertices and X)**.** If a vertex v in G has a higher degree than the function f allows ($d(v) > f(v)$), then we say that v is *unsatisfied.* The set of all unsatisfied vertices in the graph is X, or $X(G)$ if we are dealing with multiple graphs. Any vertex v in G that is not *unsatisfied* is a *satisfied* vertex.

We also have some additional definitions for components that doesn't have any unsatisfied vertices.

Definition 3.1.2 (*free*)**.** For any connected component $C \in \mathcal{C}$ with no unsatisfied vertices, let

$$\textit{free}(C) = \sum_{v \in V(C)} (f(v) - d(v))$$

be the free capacity of a component, and for any set of connected components with no unsatisfied vertices $\mathcal{D} \subseteq \mathcal{C}$

$$\textit{free}(\mathcal{D}) = \sum_{C \in \mathcal{D}} \textit{free}(C)$$

be the sum of all free capacities for the components of $\mathcal{D}$.

Definition 3.1.3 (*total*)**.** For any connected component $C \in \mathcal{C}$ let

$$\textit{total}(C) = \textit{free}(C) + 2(|E(C)| - |V(C)| + 1)$$

be the total capacity of a component, and for any set of connected components with no unsatisfied vertices $\mathcal{D} \subseteq \mathcal{C}$

$$\mathit{total}(\mathcal{D}) = \sum_{C \in \mathcal{D}} \mathit{total}(C)$$

be the sum of all total capacities for the components of $\mathcal{D}$.

The cycle edges in a component are the edges that contribute to the total capacity. They provide capacity in the sense that they can be deleted without disconnecting any components.

Definition 3.1.4 ($\mathcal{C}_{\mathit{full}}$)**.** Let $\mathcal{C}_{\mathit{full}}$ be the set of all connected components which has 0 free capacity and at least 1 total capacity. If a component C is in $\mathcal{C}_{\mathit{full}}$, we call this a *full* component.

Definition 3.1.5 ($\mathcal{C}_{\mathit{sated}}$)**.** Let $\mathcal{C}_{sated}$ be the set of all connected components that has 0 total capacity. If a component C is in $\mathcal{C}_{\mathit{sated}}$, we call this a *sated* component or tree. The reason for why a sated component must be a tree is because it has zero cycle edges (otherwise it would have some total capacity), and a connected component without any cycle edges is a tree.

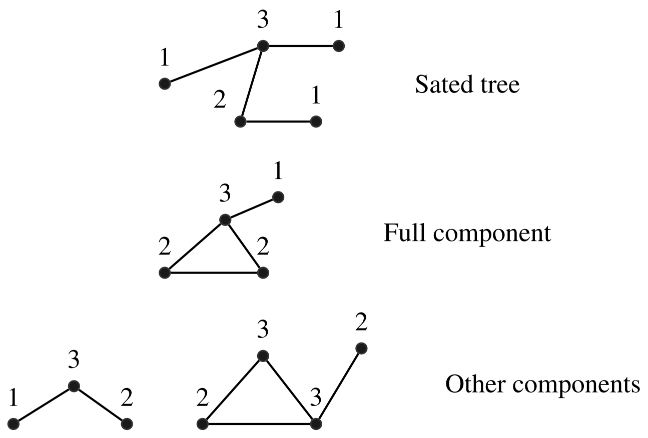

Figure 3.1: Example components when all degree constraints are satisfied (degree constraints are shown above each node)

3.2 Connection Lemma

Now we will show that if all the degree constraints in a particular instance (G, f, k) are satisfied, the question of whether the graph can be connected together or not can be solved just by looking at some basic properties of the connected components of our graph. We say that all degree constraints in the graph are satisfied if the set of unsatisfied vertices X is the empty set, or equivalently that for all vertices v in the graph, $d(v) \leq f(v)$.

3.2.1 Non-deleting connection Lemma

Lemma 3.2.1. *If every vertex v in G has $d(v) \leq f(v)$, and we have at least two connected components in the graph, then the graph can be made into a connected graph within the budget k without violating the degree constraint f using only edge additions if and only if:*

$$\forall C \in \mathcal{C} : \mathit{free}(C) \geq 1 \tag{3.1}$$

$$\mathit{free}(\mathcal{C}) \geq 2(|\mathcal{C}| - 1) \tag{3.2}$$

$$k \geq |\mathcal{C}| - 1 \tag{3.3}$$

Proof. We will first prove the forward direction of the Lemma. The proof is by induction on the size of $\mathcal{C}$.

Base case ($|\mathcal{C}| = 2$): We have that *k* is at least one, and both our components C_1, C_2 have $\mathit{free}(C_1) \geq 1$ and $\mathit{free}(C_2) \geq 1$. As $\mathit{free}(C_1) \geq 1$, there exists a vertex v in C_1 such that $d(v) - f(v) \geq 1$. By the same argument we have a vertex u in C_2 such that $d(u) - f(u) \geq 1$. If we add an edge between u and v, we join the two components, and add 1 to the degree of both u and v. Both the difference between the degree and degree limit of u and v is now at least 0, so all our degree constraints are still satisfied. We have also joined our two components, so our graph is now connected.

Inductive step ($|\mathcal{C}| > 2$): Assume that we can connect the graph G when $|\mathcal{C}| = h$, and all the conditions above are satisfied. From this assumption we must prove that we can connect the graph when $|\mathcal{C}| = h + 1$.

Let C_1, C_2 be the two components with the highest free capacity in $\mathcal{C}$. As in the previous case, we have a vertex u in C_1 and v in C_2 such that u has degree at most $f(u)-1$, and v has degree at most $f(v) - 1$. Let C' be the new component we get after connecting C_1 and C_2 by adding an edge between u and v. As we reduce both the size of $\mathcal{C}$ and k by one, and we don't connect to any vertices that does not have free connection spots, the only way we can break the conditions after connecting C' is if C' has free capacity = 0.

C' has free capacity $\mathit{free}(C_1) + \mathit{free}(C_2) - 2$ (we remove one free connection spot from each). If C' has free capacity 0, that means that both C_1 and C_2 have free capacity 1. Then all components in $\mathcal{C}$ has free capacity 1 (the minimum free capacity for each component is 1, and we selected the two components with highest free capacity). Then $\mathit{free}(\mathcal{C}) = |\mathcal{C}|$, but since $|\mathcal{C}| \geq 3$, that means condition 2 is not satisfied.

Since the forward direction of the Lemma holds for the base case and the inductive step, this holds for any number of components in the graph (as long as all the conditions in the Lemma are satisified).

We show the other direction of the implication by proving that it is impossible to connect the graph with only edge additions if we violate any of the conditions listed in the Lemma. Since the Lemma is only valid if all the degree constraints are satisfied, we keep this assumption through all the different cases.

Violating condition 3.1: Condition 1 states that for every component, we have at least one vertex v in that component which has $f(v) - d(v) \geq 1$. Targeting a contradiction, we assume that we have a component C where there is no such vertex, and we are still able to connect the entire graph using only edge additions without violating any degree constraint. To connect C to the rest of the components, we must add at least one edge to one vertex in C to some vertex in a different connected component. However this can't be done without violating a degree constraint, as for every vertex v in C the degree constraint is already tight with $d(v) = f(v)$.

Violating condition 3.2: Condition 3.2 states that the sum of the free capacity of all the components must be greater than or equal to the twice the number of components, minus two. Assume that this is not true, such that

$$\mathit{free}(\mathcal{C}) < 2(|\mathcal{C}| - 1)$$

To connect the graph, we need to add $|\mathcal{C}| - 1$ edges, as every edge addition merges at most two components. For every edge addition we must connect two vertices u, v such that we don't violate the degree constraint for either of the vertices. This means that for every pair of components C_1, C_2 we want to merge we require a vertex $u \in C_1 : d(u) < f(u)$, and a vertex $v \in C_2 : d(v) < f(v)$. Since the value of $\mathit{free}(C)$ is the sum of the difference in degree and function value for all the vertices in each component, and we add one edge between two vertices in different connected components each time we connect components, we will run out of valid vertices to add edges between if condition 3.2 is violated.

Violating condition 3.3: If condition 3.3 is violated, then $k < |\mathcal{C}| - 1$. Since we have $|\mathcal{C}|$ components, and every edge addition can at most reduce the number of components by

1, we need at least $|\mathcal{C}|-1$ edge additions to connect the graph. Therefore we can't connect the graph if $k < |\mathcal{C}| - 1$, as every edge addition uses 1 of our budget.

As we can't violate any of the conditions of the Lemma without encountering a contradiction, the reverse direction of the Lemma is proved.

□

3.2.2 General connection Lemma

The connection Lemma in the previous section gives us a useful property about the problem of connecting components using only edge additions. However we can state the necessary conditions for connecting the entire graph if all the degree constraints are already satisfied. By using this Lemma we can solve the decision problem in polynomial time if all the degree constraints are satisfied, by traversing the graph and counting the different kinds of components we have, as well as the total and free capacity of the graph.

Lemma 3.2.2 (General connection Lemma)**.** *If all degree constraints are satisfied and we have more than one component then the graph can be connected within the budget k without violating the degree constraint f if and only if:*

$$\mathit{total}(\mathcal{C}) \geq 2(|\mathcal{C}|-1) \tag{3.1}$$

$$k \geq x + (|\mathcal{C}|-1) + |\mathcal{C}_{\mathit{full}}| + 2|\mathcal{C}_{\mathit{sated}}| \tag{3.2}$$

Where

$$x = \max\left\{|\mathcal{C}| - 1 - |\mathcal{C}_{\mathit{full}}| - \lfloor\frac{\mathit{free}(\mathcal{C})}{2}\rfloor, 0\right\}$$

Proof. First we show the forward direction of the equivalence with a set of operations that reduces to a YES-instance of Lemma 3.2.1.

Base case ($|\mathcal{C}_{\mathit{sated}}| + |\mathcal{C}_{\mathit{full}}| = 0$): If no component $C_i \in \mathcal{C}$ is either sated or full, we know that every component C_i has $\mathit{free}(C_i) \geq 1$, so condition 1 of Lemma 3.2.1 is already satisfied. To satisfy condition 2 of Lemma 3.2.1, we must have $\mathit{free}(\mathcal{C}) \geq 2(|\mathcal{C}|-1)$. The remaining free capacity we need is $2(|\mathcal{C}|-1) - \mathit{free}(\mathcal{C})$.

Since we have zero of both sated and full components, the total budget k we have available is $\geq 2|\mathcal{C}| - 2 - \left\lfloor\frac{\mathit{free}(\mathcal{C})}{2}\right\rfloor$. As we need $|\mathcal{C}| - 1$ connections to connect every component by Lemma 3.2.1, we have $\geq |\mathcal{C} - 1| - \left\lfloor\frac{\mathit{free}(\mathcal{C})}{2}\right\rfloor$ budget left to gain enough free capacity. Every cycle edge that we delete increases $\mathit{free}(\mathcal{C})$ by 2, so by deleting

$|\mathcal{C} - 1| - \left\lfloor \frac{\mathit{free}(\mathcal{C})}{2} \right\rfloor$ cycle edges we have enough free capacity to apply Lemma 3.2.1. We know that we have enough cycle edges to delete, as $\mathit{total}(\mathcal{C}) \geq 2(|\mathcal{C}| - 1)$, where the contribution to total which is not from free capacity is from cycle edges. Therefore we can connect all components if $|\mathcal{C}_{\mathit{sated}}| + |\mathcal{C}_{\mathit{full}}| = 0$.

Inductive hypothesis: In the inductive step we want to show that assuming Lemma 3.2.2 holds for $j = |\mathcal{C}_{\mathit{sated}}|, l = |\mathcal{C}_{\mathit{full}}|$, then Lemma 3.2.2 also holds for the two cases

$$|\mathcal{C}_{\mathit{sated}}| = j + 1, |\mathcal{C}_{\mathit{full}}| = l \tag{1}$$

and

$$|\mathcal{C}_{\mathit{sated}}| = j, |\mathcal{C}_{\mathit{full}}| = l + 1 \tag{2}$$

When we have $j = |\mathcal{C}_{\mathit{sated}}|, l = |\mathcal{C}_{\mathit{full}}|$, the budget k we have for deletions and additions is $k \geq x + (|\mathcal{C}| - 1) + l + 2 \cdot j$.

$|\mathcal{C}_{\mathit{sated}}| = j + 1, |\mathcal{C}_{\mathit{full}}| = l$**:** By the inductive hypothesis we can connect all components when $j = |\mathcal{C}_{\mathit{sated}}|, l = |\mathcal{C}_{\mathit{full}}|$, so we need to show that by this also applies when we have one more sated component.

If we delete any edge in a sated tree, we get two components, each with free capacity of 1. Condition 1 of the Lemma still holds, as we increased total capacity by 2, and number of components by 1. Since we increased the number of components by 1, we need one more total edge addition to connect all components. The x part of our budget does not change after deleting an edge in our sated component, as we increase $\mathit{free}(\mathcal{C})$ by 2, but also increase $|\mathcal{C}|$ by 1.

After doing the above edge deletion, we have a case where $j = |\mathcal{C}_{\mathit{sated}}|, l = |\mathcal{C}_{\mathit{full}}|$. The only difference is we used one edge deletion, and we have increased our number of components by one. The extra component requires one extra edge addition compared to the case where $j = |\mathcal{C}_{\mathit{sated}}|, l = |\mathcal{C}_{\mathit{full}}|$. However we have one extra budget, as Lemma 3.2.2 gives two edits per sated component. Therefore if we can connect all components if $j = |\mathcal{C}_{\mathit{sated}}|, l = |\mathcal{C}_{\mathit{full}}|$, we can also connect all components if $j + 1 = |\mathcal{C}_{\mathit{sated}}|, l = |\mathcal{C}_{\mathit{full}}|$.

$|\mathcal{C}_{\mathit{sated}}| = j, |\mathcal{C}_{\mathit{full}}| = l + 1$**:** By assumption we can connect all components if $|\mathcal{C}_{\mathit{sated}}| = j, |\mathcal{C}_{\mathit{full}}| = l$. When $|\mathcal{C}_{\mathit{full}}| = l + 1$, we delete a cycle edge in one of the full components. Now we have a case where $|\mathcal{C}_{\mathit{sated}}| = j, |\mathcal{C}_{\mathit{full}}| = l$, so if we still satisfy condition 1 and 2 of Lemma 3.2.2 we can connect all components if $|\mathcal{C}_{\mathit{sated}}| = j, |\mathcal{C}_{\mathit{full}}| = l + 1$.

Condition 1 holds true after deleting an edge in a sated component, as we don't change either the total of $\mathcal{C}$, or the number of components.

Condition 2 also holds as the budget from x does not change (we increase free by 2 when we delete a cycle edge, and reduce $|\mathcal{C}_{full}|$ by one), and we know that we can connect all components if $|\mathcal{C}_{sated}| = j$, $|\mathcal{C}_{full}| = l$, where we have one less full component and one less k.

Since we only used one edit to go to this case, and we require one less edit, we can connect all components if $|\mathcal{C}_{sated}| = j$, $|\mathcal{C}_{full}| = l + 1$.

As we can connect all components if we have 0 of both sated and full components, and assuming we can connect if we have j sated and l full components shows that we can connect both if we have $j + 1$ or $l + 1$ of sated or full components, we can connect all components without violating the degree constraints if the conditions of Lemma 3.2.2 holds.

We now show the reverse direction by assuming that we can still connect the graph without violating the degree constraints within our budget even while violating a condition of the Lemma, and thus obtaining a contradiction.

Condition 3.1 does not hold: If condition 3.1 does not hold, then we have less total capacity for all our components than $2(|\mathcal{C}|-1)$. The relation of total capacity for the graph with the number of components never change if all degree constraints are satisfied:

If we delete a cycle edge, the total capacity of the graph does not change and we keep the same number of components. If we delete a non-cycle edge, we gain 2 total capacity, but we increase the number of components by 1. If we connect two components by adding an edge between two vertices with free capacity, we decrease the total capacity by 2, but reduce the number of components by 1. If we add an edge between two vertices with free capacity in the same component, neither the total capacity nor the number of components change.

If we have exactly one component, the total capacity must therefore be less than 0 if $total(\mathcal{C}) < 2(|\mathcal{C}| - 1)$, which means we must have a vertex with $f(v) < d(v)$. Therefore the only way to connect all components if $total(\mathcal{C}) < 2(|\mathcal{C}| - 1)$ is to violate a degree constraint, which is a contradiction.

Condition 3.2 does not hold: Assume that $k < x + (|\mathcal{C} - 1) + |\mathcal{C}_{full}| + 2|\mathcal{C}_{sated}|$, and that we can still connect all components without violating a degree constraint.

As all full components have no free capacity to connect to other components, we need to delete a cycle edge in each full component to connect them together. This uses $|\mathcal{C}_{full}|$ edits.

As every sated component can't be connected without violating a degree constraint, we need to split them up by deleting any edge in them to gain 2 components with free capacity of 1. This uses $|\mathcal{C}_{sated}|$ edge deletions, and increases the number of components by $|\mathcal{C}_{sated}|$.

We know from Lemma 3.2.1 that we need $\mathit{free}(\mathcal{C}) \geq 2(|C| - 1)$. Let $\mathcal{C}'$ denote the components after doing the deletions above. Every deletion of a cycle edge increases $\mathit{free}(\mathcal{C}')$ by 2. The remaining capacity we need is $2(|\mathcal{C}| - 1) - \mathit{free}(\mathcal{C}) - 2|C_{\mathit{full}}|$, requiring $|\mathcal{C}| - 1 - |\mathcal{C}_{\mathit{full}}| - \left\lfloor \frac{\mathit{free}(\mathcal{C})}{2} \right\rfloor$ edits. After this we must connect all components together, requiring $|\mathcal{C}'| - 1 = |\mathcal{C}| - 1 + |\mathcal{C}_{sated}|$ edge additions. This is exactly the value that k is smaller than, and we conclude that we can't connect all components without violating any degree constraints if $k < x + (|\mathcal{C}| - 1) + |\mathcal{C}_{\mathit{full}}| + 2|\mathcal{C}_{sated}|$.

□

3.3 Polynomial kernel

In this section we will provide a polynomial kernel for EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES. The general idea behind the reduction rules is from the insight provided by Lemma 3.2.2 (the general connection Lemma), which tells us that after satisfying all degree constraints only certain properties of the connected components determine whether the instance is a YES-instance or a NO-instance. Any edge deletion that satisfies degree constraints must be incident to some vertex in X, which the kernelization algorithm utilizes by compressing the graph outside of X.

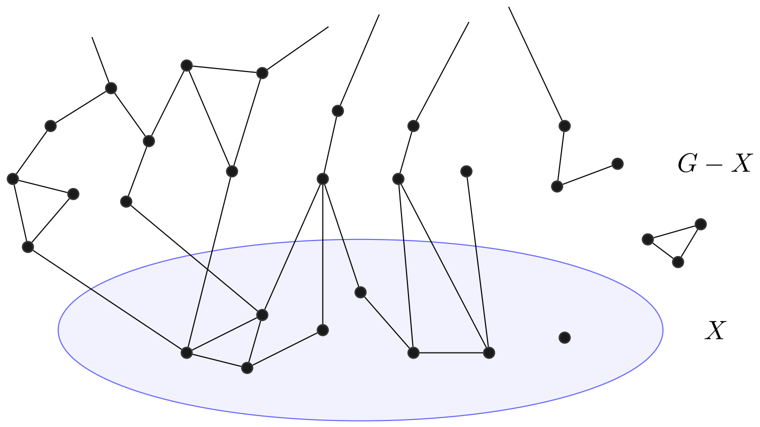

Figure 3.2: Example input graph, focused on X

3.3.1 Reduction Rules

Every reduction rule in this section takes as input an instance of EDITUBD as a tuple (G, f, k) (sometimes referred to as the *original* instance), and outputs a modified instance as the tuple (G', f', k') (sometimes referred to as the *reduced* instance).

The rules are applied in the order listed, such that in the case where two reduction rules can be applied to an instance, we always apply the first of these two rules. All of the reduction rules run in polynomial time, as they rely on identifying simple properties in the graph and then doing a bounded modification on the instance.

As we always output a reduced instance, even if we could give the answer to the problem instance, these two trivial instances are defined for convenience.

Definition 3.3.1 (Trivial NO-instance)**.** Let $V(G) = \{u, v\}$, $E(G) = \emptyset$, $f(u) = f(v) = 0$ and $k = 0$. This is a NO-instance by Lemma 3.2.2.

Definition 3.3.2 (Trivial YES-instance)**.** Let $V(G) = \{u\}$, $E(G) = \emptyset$, $f(u) = 0$ and $k = 0$ in (G, f, k). This is a YES-instance, as all degree constraints are satisfied and the graph is connected.

Reduction Rule 1 (Out of budget)**.** If

$$\frac{1}{2}\sum_{v\in X}(d(v) - f(v)) + |\mathcal{C}| - 1 > k$$

output the trivial NO-instance.

This safety of this reduction rule is guaranteed by the following Lemma.

Lemma 3.3.1. *Reduction rule 1 is safe, and runs in linear ($\mathcal{O}(|V| + |E|)$) time.*

Proof. For reduction rule 1 to be safe, we must show that if

$$\frac{1}{2}\sum_{v\in X}(d(v) - f(v)) + |\mathcal{C}| - 1 > k$$

we have a NO-instance.

Let's assume by contradiction that it is not true, i.e. even if

$$\frac{1}{2}\sum_{v\in X}(d(v) - f(v)) + |\mathcal{C}| - 1 > k$$

we have a YES-instance. Then there is a set of edge additions and deletions A such that $G\triangle A$ is a connected graph with no unsatisfied vertices, with $|A| \leq k$. Let G' be the graph obtained after all the edits of A. As A is a solution every vertex v in G' must have $d(v) \leq f(v)$. Every edge deletion reduces the degree of two vertices, so we must have at least

$$\frac{1}{2}\sum_{v\in X}(d(v)-f(v))$$

edge deletions in A. As G' is connected, and every edge addition can at most reduce the number of components with 1, we have at least $|\mathcal{C}|-1$ edge additions in A. Therefore,

$$|A| \geq \frac{1}{2}\sum_{v\in X}(d(v)-f(v)) + |\mathcal{C}| - 1 > k$$

and $|A|$ is both $\leq k$ and $> k$ from our assumption that we had a YES-instance.

□

Later reduction rules will merge vertices outside of X together to the reduce the size of the instance. We must take care not to merge certain kinds of components however, since if we merge a sated tree into a single vertex we have a single vertex with $f(v) = 0$ which immediately turns this into a NO-instance.

Reduction Rule 2 (Sated tree removal)**.** If we have a component C with no unsatisfied vertices that is a sated tree, then delete an arbitrary edge in C.

Lemma 3.3.2. *Reduction rule 2 is safe.*

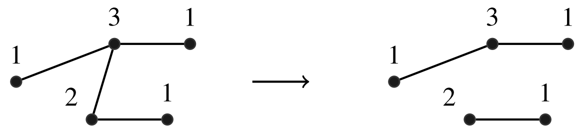

Figure 3.3: Splitting a sated tree

Proof. In Lemma 3.2.2 we show that the graph can be connected after satisfying the degree constraints if and only if conditions

$$\textit{total}(\mathcal{C}) \geq 2(|\mathcal{C}| + |\mathcal{C}_{\textit{sated}}| - 1) \tag{3.1}$$

$$k \geq x + (|\mathcal{C}| - 1) + |\mathcal{C}_{\textit{full}}| + 2|\mathcal{C}_{\textit{sated}}| \tag{3.2}$$

hold.

Assume that our original (G, f, k) is a YES-instance, and that after applying the previous reduction rule it is a NO-instance. Then we have a set A such that $G \triangle A$ is a connected graph where every vertex v has degree less than or equal to $f(v)$, and that $|A| < k$. Let $A' \subseteq A$ be an inclusion-wise minimal set of the pairs of vertices that correspond to edge deletions that satisfy degree constraints. As every vertex in C is satisfied, the component C is the same in the graph $G' = G \triangle A'$. Also, as we know that this is a YES-instance, we know that the conditions of Lemma 3.2.2 still holds. If we now delete an arbitrary edge in C in G', the first condition still holds as we add two to total, remove one sated component, and increase the number of components by two. For this to be a NO-instance we must therefore violate the second condition of the Lemma after deletion. After the deletion of an arbitrary edge in C, k is 1 smaller and we have one more component, but $\mathcal{C}_{sated}$ is one less. As the calculation for x is also the same the inequality still holds. Therefore if we have a YES-instance before deleting an arbitrary edge in C, we will still have a YES-instance after deleting an arbitrary edge in C.

For the reverse direction of the proof we must show that if we have a YES-instance after applying the reduction rule, then we had a YES-instance after applying the reduction rule. As we have a YES-instance with (G', f', k'), we have a set of edge additions and deletions that satisfies the entire graph and connects all vertices. Take A to be the minimum subset of edge deletions from the solution set that satisfies all degree constraints. A must also satisfy all degree constraints in G, as we have not altered any components adjacent to X in G'. In $G' \triangle A$ we know that since it is a YES-instance, the general connection Lemma is satisfied.

If we look at $G \triangle A$ the difference is that we have one more budget ($k = k' + 1$), and we have one more *sated* components, but two less components with a combined *total* of one. Thus if the conditions for the general connection Lemma hold in $G' \triangle A$ then they must also hold in $G \triangle A$ and we have a YES-instance in our original instance if we have it after the reduction rule is applied. □

Now we want to reduce the size of our instance, and the way we will do this is by contracting vertices outside of X that are in the same component.

Reduction Rule 3 (Component contraction)**.** If a component C in $G - X$ has more than 2 vertices, contract an uv-edge where both u and v are different vertices in C. The function f for the new vertex v' is set to $f(u) + f(v) - 2$.

Lemma 3.3.3. *Reduction rule 3 is safe.*

Proof. First we show that if we originally have a YES-instance then the contraction of u and v into v' as done in reduction rule 3 still leaves us with a YES-instance.

As our original graph is a YES-instance, there exists some set of edits A that satisfies all the degree constraints, connects the graph, and is of size $\leq k$. For any set $B \subseteq A$ the instance $(G \triangle B, k - |B|, f)$ must also be a YES-instance. If we take A' to be the minimal subset of A that satisfies all degree constraints, $(G \triangle A', k - |A'|, f)$ is a YES-instance. From Lemma 3.2.2 we know that only the *total*$(\mathcal{C})$, k and the number components, as well as the number *sated* and *full* components matter for connecting all components if all degree constraints are satisfied.

The contraction of two adjacent vertices does not change the number of components, and it can't make a non-sated or non-full component into a sated or full component. Therefore the graph obtained after contraction of two adjacent vertices outside of X must be a YES-instance if it was a YES-instance before contraction.

We will now show that if our reduced instance (G', f', k) after the application of reduction rule 3 is a YES-instance, then our original instance (G, f, k) is also a YES-instance. Since (G', f', k) is a YES-instance, we know that there exists a set of edges A, $|A| \leq k$ such that $G' \triangle A$ is a connected graph, and every vertex v in $G' \triangle A$ has $d(v) \leq f'(v)$. Take $B \subseteq A$ to be all the edges of A which does not include v'. For all edges that correspond to edge deletions in $A \setminus B$ of the form $\{v', w\}$ for some vertex $w \neq v'$, we know that either the edge $\{u, w\}$ or $\{v, w\}$ exists in G as the adjacent component contraction does not create new edges adjacent to v' other than those adjacent to either u or v. For the edge deletions that correspond to the form $\{v', v'\}$, we know that these edges must exist as either $\{u, v\}$, $\{u, u\}$ or $\{v, v\}$ in G. Therefore we can map every edge deletion in $A \setminus B$ to either an edge deletion of the corresponding edge adjacent to u, v or both. Note that all these edge deletions can't disconnect u and v, as the component contraction "removes" one edge $\{u, v\}$ from the reduced instance.

Let A' be the set of edges that maps all edge deletions incident to v' as well as all the pairs in B. $G \triangle A'$ can't have any vertices with violated degree constraints. Assume otherwise, that there exists a vertex x in $G \triangle A'$ such that $d(x) > f(x)$. Then x must have violated the degree constraint in $G' \triangle A$ as well, as we have removed the same number of edges incident to x in $G \triangle A'$ as in $G' \triangle A$.

Next, $A \setminus A'$ can contain some edge additions to connect the graph. We can take the first $f(v) - d(v)$ edge additions of the form $\{v', w\}$ and add an edge $\{v, w\}$ instead. The remaining edge additions incident to v' in $A \setminus A'$ can be mapped to $\{u, w\}$ instead. This won't violate the degree constraints of either u or v, as the degree constraints of v' is not violated in $G' \triangle A$, and $f'(v') = f(v) + f(u) - 2$. Let A^* be A' with all the remaining edge additions remapped as described above. Then $|A^*| = |A|$, and $G \triangle A^*$ must be a connected graph with no violated degree constraints: Assume to the contrary that $G \triangle A^*$

is not connected. Since the only change in the reduction is for vertices incident to u or v, there must be some vertex that is disconnected from the u, v-component. This would mean that the same vertex is disconnected from the v'-component in $G' \triangle A$.

At last, all the degree constraints in $G \triangle A^*$ must be satisfied. We have already excluded u or v from being unsatisfied, so if any degree constraints are violated it must be for some other vertex w. Then w must violate its degree constraints in $G' \triangle A$ as well, as exactly the same number of edges incident to w is added and removed, and $f(w) = f'(w)$.

Therefore if (G', f', k) is a YES-instance with the set A, (G, f, k) is also a YES-instance with the set A^*.

□

The three previous reduction rules rely on identifying simple properties in our instance graph. Therefore we can implement all these reduction rules in linear time.

Claim 1. *The conditions of reduction rules 1, 2 and 3 can be checked and performed in time* $\mathcal{O}(|V| + |E|)$.

Proof. We run a depth first search (DFS) on the input graph, where we count the number of components, the total sum of $d(v) - f(v)$ for all unsatisfied vertices, and calculate the total capacity of any component with only unsatisfied vertices. These properties allows us to determine if any of the reduction rules 1, 2 or 3 applies. All the mentioned reduction rules take linear time to implement: Outputting the trivial NO-instance in the case of 1 takes constant time. Deleting an edge in the case of 2 takes time linear to the degrees of the vertices incident to the deleted edge. Contracting an edge takes linear time in the degrees of the two vertices we contract. Therefore all these reduction rules can be checked and applied in time $\mathcal{O}(|V| + |E|)$. □

Observe that after exhaustively applying reduction rule 3 the graph $G - X$ is an independent set (even though there might be some edges, they are only self-loops). All our components of $G - X$ consists of single vertices, so we can't reduce our instance further by doing further contractions inside components. Now we will also merge separate vertices of $G - X$ if we can show that they will always belong to the same component after fixing all our degree constraints.

Reduction Rule 4 (Join components)**.** For two distinct vertices u, v in X, let $C \subseteq V(G) - X$ be the set of satisfied vertices that are adjacent to both u and v. If both u and v both have $\geq 2k + 1$ edges to vertices of C, merge two different vertices c_1 and c_2 into a new vertex c'.

The function value $f'(c')$ is set as $f(c_1) + f(c_2)$.

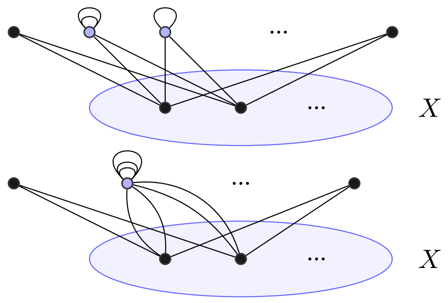

Figure 3.4: Merging two vertices of $G - X$

Lemma 3.3.4. *Reduction rule 4 is safe.*

Proof. First we will show that given that this is a YES-instance, we can always obtain a solution that keeps all of the vertices in C connected using u and v as connection points, meaning that we also have a YES-instance after application of reduction rule 4.

From Lemma 3.2.2 we know that the only properties of the graph that determines whether we can connect all components within our budget after all degree constraints are satisfied is the number of components (including the number of *full* and *sated* components) and the *total* capacity of all components. The reduction rule preserves all the original edges, so the only way we can go from a YES-instance to a NO-instance is if we alter any of these properties of the graph.

We have a YES-instance, so we have a set of of edits B such that all of G is connected without violating any of the degree constraints, and $|B| \leq k$. Since u and v are in X, B must have some edge deletions of vertices adjacent to u and v. Let $A \subseteq B$ be the minimum set of edge deletions that satisy all degree constraints in the graph, and let $A' \subseteq A$ be A except any edges between u, v and vertices of C. As (G, f, k) is a YES-instance with the set B, it still holds that $(G \triangle A, f, k - |A|)$ is a YES-instance as A is a subset of the solution B.

Let G' be the graph obtained by $G \triangle A$. From Lemma 3.2.2 we know that G' satisfies the conditions of the Lemma, since all degree constraints in G' are satisfied and we have a YES-instance. In the graph $G \triangle A'$ there is at most two vertices that violate the degree constraints, u and v. To satisfy the degree constraints of u and v in $G \triangle A'$, we must delete $d(u) - f(u)$ edges adjacent to u and $d(v) - f(v)$ edges adjacent to v.

We can satisfy the degree constraint of u by deleting $d(u) - f(u)$ edges between u and C making sure that at least one copy each of the edge $\{u, c_1\}$ and $\{u, c_2\}$ remains. Let D be the set of vertices now disconnected from u after the above deletions. We can then satisfy the degree constraint of v by deleting $d(v) - f(v)$ edges between v and $C \setminus D$, making sure to keep at least one edge $\{v, c_1\}$.

After the above deletions the edge constraints of u and v are both satisfied, and for any two vertices in $u, v \cup C$ they are adjacent to at least one of u or v, and both u and v are adjacent to c_1. Thus the entirety of $C \cup \{u, v\}$ is a connected component. The only way for this to be a NO-instance after satisfying the degree constraints of u and v is if the conditions of Lemma 3.2.2 no longer holds after merging c_1 and c_2 into one vertex. This means that either $\mathit{total}(\mathcal{C}) < 2(|\mathcal{C}| - 1)$ or $k < x + (|C| - 1) + |\mathcal{C}_{\mathit{full}}| + 2|\mathcal{C}_{\mathit{sated}}|$. Observe that $\mathit{total}(\mathcal{C})$ and $\mathit{free}(\mathcal{C})$ must remain unchanged from G', so only the number or type of components must cause this to be a NO-instance. As the deletions above ensure that we don't create more connected components, and the u, v, C-component is clearly not a sated or full component as we have deleted edges adjacent to a satisfied vertex guaranteeing a free connection spot, we still have a YES-instance after applying the rule.

We will now show that if we have a YES-instance after applying reduction rule 4, we also had a YES-instance before applying the reduction rule. The way we will do this is by obtaining a solution in our original instance by modifying the existing solution in our reduced instance.

Let (G, f, k) be our original instance and (G', f', k) be our instance after applying the reduction rule. There exists a set A of edge deletions and additions such that $G' \triangle A$ is a connected graph, $\forall v \in V(G' \triangle A) : d(v) \leq f(v)$ and $|A| \leq k$. Take $A' \subseteq A$ to be the minimum set of edge deletions that satisfies all degree constraints in G'. $(G' \triangle A', f', k - |A'|)$ must fulfill the conditions of the general connection Lemma 3.2.2, as it is a YES-instance.

Now we want to show that it is possible to edit A' into another set B such that $(G \triangle B, f, k - |B|)$ fulfills the conditions of the general connection Lemma. We know that both u and v can be in the same component as c' in $G' \triangle A'$, as both u and v has at least two edges to c' each, and we can map over the edge deletions.

For any vertex $x \notin \{u, v\}$ incident to c' that has edge deletions of the form $\{x, c'\}$ in A', this component will either have no edges incident to c' in $G' \triangle A$, or at least one edge incident. If it has no edges incident, it means x does not have to be adjacent to c' in our solution, so it also does not have to be adjacent to either c_1 or c_2 in the original graph. If it has at least one edge $\{x, c'\}$ in $G' \triangle A$, we must ensure that it is in the same component as c_1 and c_2 after deleting to satisfy edge constraints.

Thus we only need for a vertex $x \notin \{u, v\}$ that is adjacent to c' in $G' \triangle A'$ to be adjacent to either c_1 or c_2, as the vertices u, v link them together into one connected component. To satisfy x's degree constraint we can delete as many edges of the form $\{x, c_1\}$ or $\{x, c_2\}$ as there are edges $\{x, c'\}$ in A'. We know that there are at least $2k + 1$ edges between c_1 or

c_2 and u or v, so it is always possible to delete enough edges without disconnecting any of the vertices c_1, c_2, u or v as we can have at most k deletions. This way we do not create components in our graph, we use the same number of deletions as we would in $G' \triangle A'$, and we fix the same amount of degree violations. Then for any vertex $x \notin \{u, v\}$ it is possible to remap the deletions and still maintain a YES-instance. □

The reduction rule 4 requires that we for every pair of vertices in X identify the vertices of $G - X$ that are adjacent to both of these vertices.

Claim 2. *Reduction rule 4 can be done in time $\mathcal{O}(|V|^2 \cdot (|V| + |E|))$.*

Proof. For any pair of vertices v, u in X, we can check how many edges each have to common vertices of $G - X$ by running a graph search from u and v, identifying the vertices they are both adjacent to, and after that count the number of edges. As we do this for every pair of vertices in X, this must be done $\binom{|X|}{2}$ times. A graph search takes time $\mathcal{O}(|V| + |E|)$, so an upper bound the time required for identifying the number of edges to common vertices of all pairs of X is $\mathcal{O}(|V|^2 \cdot (|V| + |E|)$. Merging together two vertices takes time proportional to the degrees of these vertices, so the total running time of checking if this reduction rule applies and running it is bounded by $\mathcal{O}(|V|^2 \cdot (|V| + |E|))$. □

The previous rule for contractions reduce the number of vertices in $G - X$ that are adjacent to at least two vertices of X. However we have no bound on the number of vertices that are adajcent to only one component of X, so we want to limit this kind of component as well. We do this by utilizing the fact that we can only interact with a limited number of such components when editing according to our budget, so if we keep the best ones to do edits with in this step we can discard the ones that we never have to do edits with if we have a solution.

Reduction Rule 5 (Irrelevant component removal). For a vertex $x \in X$, let $C \subseteq V(G) - X$ be the set of vertices that are incident to only x. If $|C| > \frac{k}{2} + (f(x) - d(x))$, let $C_{\textit{edges}}$ be the first $f(x) - d(x)$ vertices in C sorted in descending order on *free* capacity and $C_{\textit{loops}}$ be the $\lfloor \frac{k}{2} \rfloor$ first members of $C \setminus C_{\textit{edges}}$ sorted on highest number of self-loops to fewest.

Remove a vertex c in $C \setminus (C_{\textit{edges}} \cup C_{\textit{loops}})$ and reduce $f(x)$ by the number of $\{x, c\}$-edges in $E(G)$, such that $d_{G'}(x) - f'(x) = d_G(x) - f(x)$.

Figure 3.5 illustrates a case when reduction rule 5 can apply.

Lemma 3.3.5. *Reduction rule 5 is safe.*

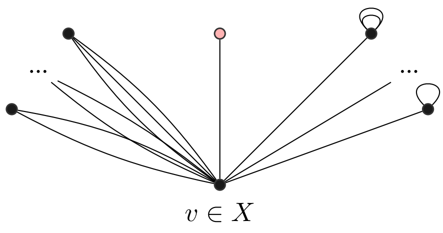

Figure 3.5: A vertex in $G - X$ (colored red here) that can be removed

Proof. We have to show that we have a YES-instance before applying reduction rule 5 if and only if the instance after applying the reduction rule is a YES-instance. First we will show the forward direction of the equivalence. Assume that we have a YES-instance before applying reduction rule 5. Then we have a set of edge deletions and additions A such that $G \triangle A$ is connected, $\forall v \in V(G \triangle A) : f(v) \geq d(v)$, and $|A| \leq k$. Targeting a contradiction we assume that after removing c from C according to the reduction rule that we obtain a NO-instance.

Take $A' \subseteq A$ to be the edge deletions in A that satisfies all unsatisfied vertices in X except any edges in adjacent to C, then $(G' = G \triangle A', f, k - |A'|)$ is also a YES-instance. The only vertex maybe not satisfied in G' is x, and x can be satisfied by only deleting edges between x and vertices of C. Since we now have a NO-instance, removing c from G must either cause us to create more components when satisfying x, or reduce *total*$(\mathcal{C})$ below the threshold $2(|\mathcal{C}| - 1)$, or creates more sated or full components after all degree constraints are satisfied.

Removing c can't cause us to create more components when satisfying x than before removing c, as we create more components when we fully disconnect a component from x, and we keep the $d(x) - f(x)$ components in C with the most multiedges to x. Then for this to be a NO-instance we must either reduce *total*$(\mathcal{C})$ below the threshold, or increase the number of sated or full components. As we keep the $\left\lfloor \frac{k}{2} \right\rfloor$ components in C with the highest number of self-loops, we are guaranteed to keep enough cycle edges to ensure that the total capacity of our components have the maximum contribution from x. Finally, we see that we never create more sated or full components when deleting edges to satisfy x, as all the remaining edge deletions must create at least one free connection spot each.

We will now show the reverse direction of the equivalence. If we have a YES-instance after applying reduction rule 5, then we have a set A of set of edge additions and deletions that connects the entire graph and satisfies all degree constraints in G'. The same set of edge additions and removals must also satisfy the graph G, as c is part of the same connected component as x so it won't be disconnected, and the extra edges that c contribute to the degree limit of x are already accounted for as $d_G(x) - f(x) = d_{G'}(x) - f'(x)$. □

Reduction rule 5 requires that we for each vertex in X identify the vertices of $G - X$ that are adjacent to only this vertex, and sorting them based on number of multiedges or the number of self-loops.

Claim 3. *Reduction rule 5 can be done in time $\mathcal{O}(|V| \cdot (|E| + |V| \log |V|))$.*

Proof. For each vertex v in X we identify the vertices that are adjacent to v only, which can be done by scanning through the edge set E. We then do two sorts on the vertices of $G - X$ adjacent to v only, and sorting a list of n elements can be done in time $\mathcal{O}(n \log n)$ with for example merge sort. Removing a vertex from the graph can be done in linear time. Therefore the running time of identifying if reduction rule 5 applies and performing it is bounded by $\mathcal{O}(|V| \cdot (|E| + |V| \log |V|))$. □

We have now bounded the number of components in our graph, so we will now bound the maximum number of edges in our graph. We do this by observing that if we have more copies of an edge than what can possibly be deleted, we only need to keep a certain number of these edges. The same argumentation holds for any self-loops in our graph.

Reduction Rule 6 (Irrelevant edge removal)**.** If we have a multiedge of multiplicity $\geq k + 1$ remove one of these edges such that the multiplicity is reduced by one. If this was a non-self-loop between some vertices u and v set $f'(u) = f(u) - 1$ and $f'(v) = f(v) - 1$. For self-loops on a vertex u we set $f'(u) = f(u) - 2$.

Lemma 3.3.6. *Reduction rule 6 is safe, and can be done in time $\mathcal{O}(E)$.*

Proof. We have to show that we have a YES-instance before applying reduction rule 6 if and only if the instance after applying the reduction rule is a YES-instance.

First we will show the forward direction of the equivalence. As (G, f, k) is a YES-instance, we know that there exists a set of edge additions and deletions A that satisfies all the degree constraints and connects G, and $|A| \leq k$. If we take A' to be the pairs corresponding to edge deletions in A that contribute to satisfying a degree constraint, we know that Lemma 3.2.2 returns YES for the instance $(G \triangle A', f, k - |A'|)$.

All degree constraints are satisfied in $G' \triangle A'$: Assume otherwise, that we have a vertex x such that $f(x) < d(x)$ in $G' \triangle A'$, but $f(x) \geq d(x)$ in $G \triangle A'$. Then either the number of edges incident to $x \in G'$ removed by A' is less than the number of edges incident to $x \in G$ removed by A', or the difference between the degree of x and the degree constraints $f(x)$

is bigger in G' than in G. As A' can remove at most k edges, and we only remove an edge from G to G' if the multiplicity of a multiedge is more than $k+1$, we must remove the same amount of edges incident to x in both G and G'. The difference between the number of edges incident to x and the degree constraint of x cannot be different in G and G', as when we remove one edge (or two in the case of a self-loop) incident to x we reduce the degree constraint of x by one (or two in the case of a self-loop). Thus all degree constraints are satsified in $G' \triangle A'$, and we are able to apply Lemma 3.2.2 to obtain a solution for our instance. Assume for a contradiction that $(G' \triangle A', f', k - |A'|)$ is a NO-instance.

As the number of edges that must be deleted incident to each unsatisfied vertex is unchanged in G', and we can always delete as many of one edge in G' as in G, we must break one of conditions 3.1 or 3.2 for it to be a NO-instance. We can never have more components in $G' \triangle A'$ than in $G \triangle A'$, as we can remove at most k edges between two vertices, and any edge that exists in G but not in G' must be a multi-edge with multiplicity of at least $k+1$. Then if condition 3.1 does not hold it must be because *total*$(G' \triangle A')$ is less than what is needed compared to the number of connected components in $G' \triangle A'$. We know that $G \triangle A'$ has at most $k+1-|A'|$ components due to $G \triangle A'$ being a YES-instance with budget $k - |A'|$, so $G' \triangle A'$ also has at most $k+1-|A'|$ components. For 3.1 to be violated, we must then have that *total*$(G' \triangle A') < 2(k - |A'|)$. As we have at least $k+1$ uv-edges in G', we will have at least $k+1-|A'|$ uv-edges left in $G' \triangle A'$. Since uv-edges past the first are cycle edges, we know that the *total* capacity of G' must be at least two times the number of uv-edges minus one, or $\geq 2(k - |A'|)$. This contradicts the fact that 3.1 is violated for our instance.

Then for $(G' \triangle A', f', k - |A'|)$ to be a NO-instance, 3.2 must be violated. We already know that the number of components in $G \triangle A'$ and $G' \triangle A'$ is the same, so we can only violate 3.2 if we have more sated or full components in $G' \triangle A'$ than in $G \triangle A'$, or we have less *free* capacity in $G' \triangle A'$ than in $G \triangle A'$. As we have at least one cycle edge left in the component where we removed an edge in this reduction, we can't have more *sated* trees in $G' \triangle A'$ than in $G \triangle A'$ because the component of u and v will never be a *sated* tree. We also can't have more *full* components in $G' \triangle A'$, as the only component we alter compared to $G \triangle A'$ is the component of u and v, and the relative difference between each vertex's degree constraint and number of incident edges is the same in both G and G'. For the same reason the *free* capacity of $G' \triangle A'$ must be the same as the *free* capacity of $G \triangle A'$.

We see that there is no way that $(G' \triangle A', f', k - |A'|)$ is a NO-instance if $(G \triangle A', f, k - |A'|)$ is a YES-instance, which concludes the proof of the forward direction.

We will now show the reverse direction of the equivalence by showing that given that (G', f', k) is a YES-instance, then (G, f, k) must be a YES-instance. As our reduced instance is a YES-instance, we have a set minimum set of edge additions and deletions that connects the entire graph and satisfies all degree constraints A, which is of size at most k. This set A must also satisfy all degree constraints in G': Assume otherwise, that $G \triangle A$ has a vertex x which is not satisfied. As the difference in the degree constraint and number of edges incident for x is the same in both G and G', then A must remove fewer edges or add more edges incident to x in G than in G'. If A removes fewer edges, then this must be because some edge in A does not exist in G – but as $E(G') \subseteq E(G)$ this can't happen. If A can't add any more edges in G than in G', as the only edge that differs between G and G' is the uv-edge, which won't be added multiples of as u and v already are in the same component and A is minimal.

As all degree constraints are satisfied in $G \triangle A$, the only way that A is not a solution to (G, f, k) is if we have more than one component in $G \triangle A$. Assume that this is the case, that we have at least two different connected components such that there is no path between some pair of vertices c_1 and c_2 in $G \triangle A$, but there is a path between c_1 and c_2 in $G' \triangle A$. Then all the edges of the path between c_1 and c_2 can't exist in $G \triangle A$, but as $E(G') \subseteq E(G)$ we know that all the edges in $G' \triangle A$ must also exist in $G \triangle A$.

Since A satisfies all degree constraints in $G \triangle A$, the graph $G \triangle A$ is connected and $|A| \leq k$, (G, f, k) must be a YES-instance if (G', f', k) is a YES-instance.

For the running time of this reduction, it is easy to see that we can see if this reduction rule applies by scanning through the set of edges and counting the number of edges that map to the same pair of vertices, which takes time $\mathcal{O}(|E|)$. Removing an edge and adjusting the degree constraint of the incident vertices can be done within the same time bound. □

The one remaining thing we can't bound by any of the reduction rules is the maximum size of the degree constraint of any vertex. This last reduction rule gives us an upper bound on the maximum degree constraint for any vertex dependent on k only.

Reduction Rule 7. If a vertex v in G has $f(v) > |V(G)| \cdot (k+1)$, set $f'(v) = |V(G)| \cdot (k+1)$.

Lemma 3.3.7. *Reduction rule 7 is safe, and can be done in time $\mathcal{O}(|V|)$.*

Proof. Assume that (G, f, k) is a YES-instance, and that (G, f', k) is a NO-instance. The maximum degree any vertex can have in our graph is $|V(G)| \cdot (k+1)$, as we have at most $|V(G)|$ distinct multiedegs, and reduction rule 6 ensures that the maximum multiplicity of any multiedge is $k+1$. Thus the reduction rule won't make satisfied vertex unsatisfied, as v has to be satisfied under both degree constraints.

The maximum number of edge additions incident to v in any solution is k, as this is the size of our budget. Assuming that we have a solution of minimal size the only reason to add edges incident to v is to connect components to v. Therefore v can't have any edges to some other vertex u if a solution requires edge additions incident to v. If this is the case then the maximum number of distinct multiedges incident to v in G must be $\leq |V(G)|-1$, and then the maximum degree of v is at most $\leq (|V(G)|-1)(k+1)$ due to reduction rule 6. Therefore any case when a minimal solution adds edges incident to v as part of the solution for (G, f, k), the reduction in the degree constraint for v will never cause v to be unsatisfied with the same set of edge additions or deletions in (G, f', k), and our assumption that (G, f', k) is a NO-instance can't be true.

For the reverse direction it is easy to see that any solution for (G, f', k) must also work for (G, f, k) as the instances are the same except for a more lax degree constraint in f compared to f'.

Reduction rule 7 can be done in time $|V|$ by simply scanning through the function values for every vertex, and setting it lower if it exceeds the bound. $\square$

If no reduction rule applies we have our fully reduced instance. We will now bound the size of this final instance, to show that the reduction rules constitute a kernel.

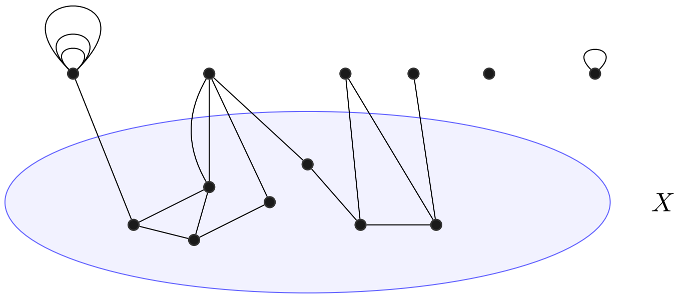

Figure 3.6: How a final instance could look like

3.3.2 Analysis

The running time of every reduction rule is polynomial. Every reduction rule reduces either the number of edges or the number of vertices except reduction rule 7, which can be done in time $\mathcal{O}(|V(G)|)$. As a consequence the maximum number of reduction rules we can apply are bounded by a polynomial depending on the size of the input graph, and combining this with the fact that every reduction rule takes time polynomial in the size of the instance the kernelization algorithm takes polynomial time.

To give a bound on the number of vertices and edges in our reduced instance, we must observe the effects of exhaustively applying all the reduction rules.

Bounding the number of vertices

The bound on the number of vertices is obtained by looking at four different kinds of vertices we can have in the graph after no more reductions can take place, and bounding the number of each kind of vertex. In our reduced instance every vertex is either in X, disconnected from X, adjacent to only one vertex in X or adjacent to two or more vertices of X. Note that since every component of $G - X$ is a single vertex due to reduction rule 3, we will refer to the components of $G - X$ as vertices here.

The number of vertices in X is bounded by $2k$, since each vertex in X needs at least one edge incident to it removed, and we can at most remove k edges. Reduction rule 1 ensures that we reject instances where $|X|$ is too big.

The number of vertices that can be disconnected from X is bounded by $k + 1$, the total number of components we have in the graph. Since we can connect at most $k + 1$ components by edge additions, reduction rule 1 rejects instances where we have too many components.

For vertices outside of X that are adjacent to only one vertex of X we know that we have at most $2k$ per vertex in X, since otherwise reduction rule 5 would apply.

The number of vertices outside of X adjacent to two or more vertices of X is bounded from the fact that for any pair of vertices of X, there are at most $2k + 1$ different vertices that are adjacent to these two vertices. This is due to reduction rule 4, which joins vertices together if there exists two vertices of X that have more than $2k + 1$ combined edges to the same set of vertices outside of X.

The number of vertices is therefore bounded by

$$2k \cdot 2k + 4k^2 \cdot (2k + 1) + k + 1 = 8k^3 + 8k^2 + k + 1$$

which is $\mathcal{O}(k^3)$ vertices.

Bounding the number of edges

In our reduced instance all edges are either self-loops, internal in X or crossing between X and $G - X$ (the only edges of $G - X$ are self-loops). The maximum number of self-loops any vertex v can have is $k + 1$, as otherwise reduction rule 6 would apply. As our reduced instance has at most $8k^3 + 8k^2 + k + 1$ vertices, we have at most

$$(k+1) \cdot (8k^3 + 8k^2 + k + 1) = 8k^4 + 16k^3 + 9k^2 + 2k + 1$$

self-loops in our reduced instance.

The number of different multiedges internal in X is $\binom{|X|}{2}$, and as the multiplicity of any multiedge is at most $k+1$, the maximum total number of edges internal in X is

$$\binom{|X|}{2} \cdot (k+1) \leq \binom{2k}{2} \cdot (k+1) = k(2k-1)(k+1) = 2k^3 + k^2 - k$$

For every pair of vertices in X we have at most $2k+1$ vertices of $G - X$ that are adjacent to those two vertices of X, as otherwise reduction rule 4 would apply. The maximum number of edges we can have in such a case is if one of the two vertices of X always has the maximum $k+1$ edges to each of these vertices of $G - X$, while the other vertex of X only has one edge to each of these common vertices (as otherwise reduction rule 4 would again apply). This gives a maximum of

$$\binom{|X|}{2} \cdot (2k+1)(k+2) \leq k(2k-1)(2k+1)(k+2) =$$

$$(k^2 + 2k)(4k^2 - 1) = 8k^4 + 4k^3 - k^2 - 2k$$

edges to vertices of $G - X$ that is adjacent to at least two vertices in X.

Every vertex in X has at most $2k$ vertices in $G - X$ which is only adjacent to that vertex, and the maximum number of edges between a vertex in X and a private component is at most $k+1$, giving an upper bound of $2k \cdot (k+1)$ edges between private components and vertices in X.

In summary we have at most

$$8k^4 + 16k^3 + 9k^2 + 2k + 1 + 2k^3 + k^2 - k + 8k^4 + 4k^3 - k^2 - 2k =$$

$$16k^4 + 22k^3 + 9k^2 - k + 1$$

edges in our reduced instance, or $\mathcal{O}(k^4)$ edges in total.

Theorem 3.3.8. EDITUBD *has a polynomial kernel with $\mathcal{O}(k^3)$ vertices and $\mathcal{O}(k^4)$ edges, with at most $2k$ vertices in the reduced graph being unsatisified (members of the set*

X). The graph induced on the satisfied vertices is an independent set, and the maximum degree constraint f for any vertex is $\mathcal{O}(k^4)$.

Proof. The previous bounding of the number of vertices and edges gives us the maximum size bound for our reduced instance. The fact that reduction rule 3 does not apply guarantees that the graph induced on the vertices that are not satisfied is an independent set.

The bound on the maximum degree constraint comes from reduction rule 7, as the maximum degree contraint for any vertex is $|V(G)| \cdot (k+1)$ and $|V(G)| \in \mathcal{O}(k^3)$ then the maximum degree constraint for any vertex is $\mathcal{O}(k^4)$. □

From the kernelization algorithm we also obtain a FPT algorithm that exploits the bounded number of edges that can contribute to satisfying degree constraints, which we combine with the general connection Lemma to check if the graph can be connected after satisfying all degree constraints.

Theorem 3.3.9. EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES *has a FPT algorithm with running time* $2^{\mathcal{O}(k \log k)} \cdot n^{\mathcal{O}(1)}$

Proof. After exhaustively applying the above reduction rules, the maximum number of edges we can have in an instance is bounded by $16k^4 + 22k^3 + 9k^2 - k + 1$. By running a brute force deletion of at most k of these edges, checking if all degree constraints are satisfied and if so checking if the graph can be connected using the general connection Lemma (3.2.2), we have a

$$(16k^4 + 22k^3 + 9k^2 - k + 1)^k \cdot n^{\mathcal{O}(1)} \in 2^{\mathcal{O}(k \log k)} \cdot n^{\mathcal{O}(1)}$$

algorithm for EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES. □

3.4 Single exponential algorithm

After running our kernelization algorithm, we will have a graph with a very particular structure: The components of $G - X$ are single vertices with some potential self-loops. As we have no edges from a vertex of $G - X$ to another of $G - X$, all connectivity runs through the vertices of X. Additionally the size of X is at most $2k$.

We now want to solve the rest of the problem, which we will do by trying to find the set of edge deletions needed to satisfy all the degree constraints in our graph. If we find any set that satisfies the degree constraints we can just make use of the general connection Lemma to give the answer to our instance in polynomial time.

The approach we will take for finding this set of edges to delete is to use the structure in the graph obtained after kernelization in combination with a random coloring of edges in the graph to guess how the vertices of X are separated after the edges that satisfy degree constraints in a YES-instance are deleted. The technique

3.4.1 X-SUPPORTING SUBGRAPH

We want to show that due to the structure of the reduced instance after the kernelization, there exists a edge-induced subgraph of size linear to X that preserves both connectivity and edge-biconnectivity between vertices of X. The structural properties of the post-kernelization graph we exploit is the fact that we can separate the graph into two sets of vertices $U = X$ and $I = V(G) - X$, such that I is an independent set.

If we have such a structure, the following Lemma guarantees the existence of a U-supporting subgraph of linear size. We will denote a subgraph that preserves connectivity and edge-biconnectivity for any set of vertices A to be an *A-supporting subgraph.*

Lemma 3.4.1 (U-supporting subgraph of linear size)**.** *For any $G = (U \cup I, E)$, where $|U| \geq 1$ and I is an independent set there exists a subgraph $H \subseteq G$ of at most $5(|U| - 1)$ edges such that if $u, v \in U$ are connected or edge-biconnected in G they are also connected or edge-biconnected in H.*

The way we will show this Lemma is by constructing the subgraph H, and then proving that this construction means that H is a U-supporting subgraph.

Construction of H

Find a set of edges constituting a spanning forest of G, and remove all edges incident to vertices in I that have degree 1 in this spanning forest obtaining a set of edges F. Add all vertices in I which have edges incident in the set of edges F to U', $U' = V(G[F]) \cup U$, and let $I' = I - U'$. Let G' be the graph $G - F$.

Lemma 3.4.2. *The spanning forest F uses at most $2|U| - 2$ edges.*

Proof. We will show the size of the spanning forest by induction on the size of U. In the base case of $|U| = 1$ the only edges that are found in the original spanning forest are edges between the one vertex of U and vertices in I. As we don't add any cycle edges, the maximum degree of any vertex of I in this spanning forest is 1, and the spanning forest F will have 0 edges in total. Since $2|U| - 2 = 2 \cdot 1 - 2 = 0$ the base case holds.

For the inductive step we assume that if $|U|$ is k, then the set of edges F is of size $\leq 2k - 2$. We must now show that given this assumption, if the size of U is $k + 1$ then the size of F is at most $2(k + 1) - 2$.

When the size of U is $k + 1$, we find the initial spanning forest and set of edges F as usual. If we remove a leaf vertex u of U from the graph $G[F]$, and again remove any edges to vertices in I that has degree 1, we know that this set of edges F' must be of size at most $2k - 2$ due to our inductive hypothesis. We can extend this set of edges to include u using at most two edges, as u is a leaf in $G[F]$, and the shortest path from u to a U-vertex must be at most two edges long as I is an independent set.

In conclusion the size bound for F of $2(|U| - 1)$ is proven by induction on the size of the set U. □

Now, do the same for the graph G'. Find a spanning forest in G', and let F' be the edges of this spanning forest without any edges to vertices of I' that has degree one in the spanning forest of G'.

Lemma 3.4.3. *The spanning forest F' uses at most $3(|U| - 1)$ edges.*

Proof. Again we will use induction on the size of U to prove the size of the spanning forest.

For the base case when $|U| = 1$, the only edges that are found in the original spanning forest are edges between the one vertex of U and vertices in I. Since we remove all vertices of I with degree 1, none of the edges will be added to the final set of edges F', and $|F'| = 0 \leq 3(|U| - 1) = 0$.

In the inductive step we assume that if the size of U is k, then the size of the set of edges F' is at most $3(k - 1)$. We will now show that with this assumption we can show that if the size of U is $k + 1$ then the size of the resulting set of edges F' is at most $3(k + 1 - 1) = 3k$.

If we consider the graph $G - u$ where $u \in U$ is some leaf vertex in the graph $G[F']$ we now have a graph where the size of $U = k$, so here we will find a set of edges F'' of size at most $3(k - 1)$ by our inductive hypothesis. Adding u back into the graph will extend the size of U by one, and the size of U' by at most 2. This is because we only add vertices from I to U' if they have degree at least two in the spanning forest of G. The maximum number of edges in a spanning forest is $|V| - 1$, so increasing $|V|$ by one we have at most one more edge in the initial spanning forest. If we have an additional edge in the spanning forest it must be an edge incident to u and some other vertex. Thus at most one vertex different from u will have a higher degree, so at most one more vertex can be added to U'. Assume that i is a vertex in I that is added to U' when u gets added back into the graph.

Connecting i back to the graph $G'[F'']$ takes at most one edge, as i was originally a part of I. Connecting u back to the graph $G'[F'']$ takes at most two edges, as u was a leaf in $G'[F']$, and the shortest path from u to any vertex in U' is at most 2 edges long.

As adding u back into the graph F' would be extended by at most 3 edges we have shown the size bound for F' by induction. □

Let F^* be the combined set of edges from F and F', $F^* = F \cup F'$. Let H be the edge-induced subgraph of G with edge set F^*. Since $|F^*|$ is bounded by $5(|U| - 1)$ H has $\leq 5(|U| - 1)$ edges. Now we must show that H is an U-supporting subgraph.

Claim 4. *$u, v \in U \subseteq G$ are edge-(bi)connected in G if and only if u, v are edge-(bi)connected in H.*

Proof. We will first prove the statement for connectivity, and then for edge-biconnectivity.

$u, v \in U$ **are connected in** G **if and only if** u, v **are connected in** H

First, observe that since u and v are in the set U and are connected in G, they are also in the subgraph H. Targeting a contradiction, we assume that u, v are connected in G, but are not connected in H. As u, v are connected in G, there exists a u, v-path in G. Then u and v must belong to the same connected component in the spanning forest of G. All the edges in the spanning forest of G except edges to leaves outside of U were added. Then an edge of the u, v-path in G must be one of the leaf edges – but this cannot be true as both u, v are in U and a u, v-path can't go through a non-related leaf in the spanning forest. Thus if u and v are connected in G, they must be connected in H.

For the reverse direction, we know that u, v are connected in H, so there exists a $u, v-$path in H. As $H \subseteq G$ the same path must exist in G, and u and v are therefore connected in G if they are connected in H.

$u, v \in U$ **are edge-biconnected in** G **if and only if** u, v **are edge-biconnected in** H

We prove the forward direction first: Assume that even though u, v are edge-biconnected in G, there exists an edge $e \in H$ such that $H - e$ causes u, v to no longer be connected. Since e was added to H, it must be part of the first or second spanning forest in G.

If it was part of the second spanning forest, then deleting e can't possibly disconnect u and v, as they are connected with only the edges of the first spanning forest if they are connected in G. Therefore e must be added as part of first spanning forest. Since removing e from H disconnects u and v, e must be an edge of a u, v-path in H. As u

and v are edge-biconnected in G, they must be connected in $G - e$. Then there exists a u, v-path in $G - e$.

If we look at the vertices of the u, v-path in $G - e$ it must at one point leave the component of u in H, and at some point enter the component of v in H. Let u' be the last vertex in U' of the u, v-path that is in the component of u in $H - e$, and v' be the first vertex of U' in the u, v-path that enters the component of v in $H - e$. We might add some edges of the u', v'-path to H as part of the first spanning forest F, meaning that u' and v' might not be in the same connected component in $G - F$. However, every vertex incident to any edge added by F is a member of U'. Any edge $\{a, b\}$ of the u', v'-path not added by F would cause a and b to be in the same connected component in the spanning forest of $G - F$, and as a and b are in U' the edges of F' must cause a and b to be connected in $H - e$. Then for every edge of the u', v'-path in $G - e$ this edge was either added to H by the first spanning forest, or the edge is between two vertices of U' that are in the same connected spanning forest in $G - F$, and thus the vertices must also be connected with the edges of F' which are added to H. Note that e can't be any of these edges, as the u', v'-path exists in $G - e$, and e was added as in the first set of edges F. Thus we can construct another path from u to v if e is deleted by taking the path from u to u', then from u' to v' where for every pair of vertices in the u', v'-path in G there must exist some path between them in $H - e$ and lastly from v' to v, even if e is deleted.

As we know that if any edge e is removed from H and u and v are two biconnected vertices in G, they would still be connected in H. Therefore u and v are edge-biconnected in H if they are edge-biconnected in G.

For the reverse direction, we know that u, v are edge-biconnected in H. As H is a subgraph of G we can never find a single edge to remove that disconnects u and v in G if they are edge-biconnected in H. □

Thus we have a linear sized U-supporting subgraph, H. If we take the graph that we obtain after exhaustively applying all the reduction rules of the kernel, X will also have a X-supporting subgraph of size $\leq 5(|X| - 1) \leq 5(2k - 1) = 10k - 5$ edges.

3.4.2 Solution skeleton

Now that we have shown that we can find a X-supporting subgraph of linear size, we will now define a solution skeleton for a YES-instance. To make reduction rules easier we will define this solution skeleton for a subset of the solution A^* for any YES-instance which *only* satisfies degree constraints and disconnects the least amount of components of any such subset of any valid solution.

If the instance (G, f, k) is YES-instance, we know that there exists some family of sets of edge deletions that satisfies all degree constraints and does not change the graph into a NO-instance if all the deletions are applied ($\mathcal{A} = \{A \subseteq E(G) \mid (G' = G \triangle A, f, k - |A|)$ is a YES-instance $\wedge\ X(G') = \emptyset\}$). We take $A^* \in \mathcal{A}$ to be the smallest set of edges that gives the minimum number of components in G' over all sets in $\mathcal{A}$.

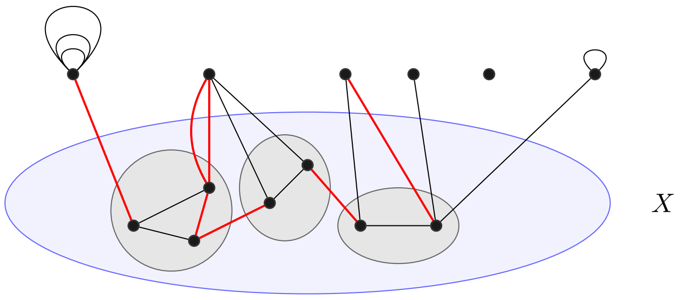

Figure 3.7: An example set of edge deletions that satisfies all degree constraints, in red

In figure 3.7 we can observe that the vertices of $X(G)$ are separated into some components, marked with the black circles.

If we apply the deletions of A^* to our graph G, we will end up with a graph where every vertex has satisfied its degree constraint. The issue is that we do not know the set A^* for our instance. To work around this we will instead show that we can reobtain the set A^* if we have some extra information about which edges we definitly can't delete, and which edges might be deleted. The information about what edges we can't delete we denote as a *solution skeleton* for our YES-instance. Obviously the solution skeleton only makes sense if we are dealing with a YES-instance.

In the graph $G' = G \triangle A^*$ we know that a $X(G)$-supporting subgraph with at most $10k - 5 < 10k$ edges exist. The edges of this subgraph are part of the solution skeleton for G.

Additionally the solution skeleton needs to encode some extra information about the vertices of $G - X$ which has some incident edges in A^*. If any such vertex has any incident edges that aren't self-loops in $G \triangle A^*$, we require that any one of these edges are in the solution skeleton. The maximum number of edges required to be in the solution skeleton by this rule is at most k, as we can delete at most k edges and the maximum required by this rule is if every edge deletion is an edge incident to k different vertices of $G - X$.

Thus the size of the solution skeleton is at most $10k - 5 + k < 11k$ edges. The maximum size of A^* in the graph is k, as that is our budget. If we have correctly encoded the information that the edges of the solution skeleton must never be deleted, as well as

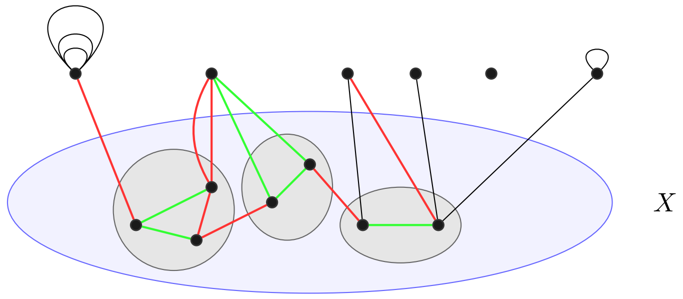

Figure 3.8: The edges of the X-supporting subgraph colored in green, and the solution edges colored red.

the information that the edges of the solution A^* can be deleted, we will show that we can delete a set of edges that satisfies all degree constraints and maintains that the resulting graph and remaining budget still is a YES-instance.

To encode the information needed for this we introduce a coloring of the edges in our instance. Every edge will be colored either *red* or *green*, where red edges may be deleted, and green edges can never be deleted. If all the edges of the solution skeleton of A^* are colored green and all the edges of A^* are colored red we have encoded the information about our solution and solution skeleton that we wished to encode, and way say that this is a *correct* coloring. Since the number of edges that must be colored correctly is less than $11k + k = 12k$ we can randomly color every edge either red or green with equal probability and obtain a correct coloring with probability $\geq \frac{1}{2^{12k}}$.

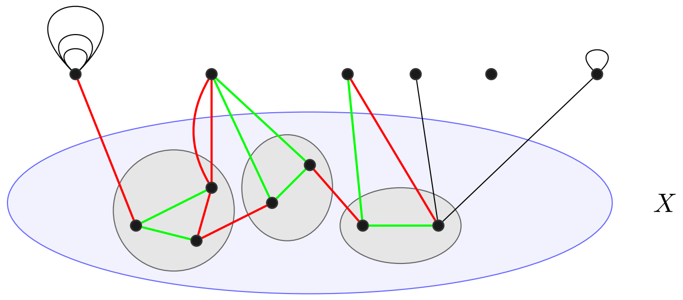

Figure 3.9: A coloring of the edges of an example solution (red), and the corresponding solution skeleton (green)

Edges that are colored black in figure 3.9 indicate edges that aren't a part of the solution or the corresponding solution skeleton, and can therefore be colored either red or green.

3.4.3 Reobtaining the solution

To make matters more explicit for this section we define the combination of our previous instance and a given coloring as a new problem. We define this problem to answer YES if the instance of EDITUBD is a YES-instance *and* the coloring c correctly colors the edges of the solution A^* red and the associated solution skeleton with A^* green. Any edge not in A^* or the corresponding solution skeleton can have any color. This allows us to construct an algorithm using reduction rules that are safe due to preserving the instance. Note that we also assume that the graph G has the properties guaranteed by the kernelization algorithm (Theorem 3.3.8).

CORRECTLY COLORED EDITUBD

Input: A graph $G = (V, E)$, functions $f : V(G) \to \mathbb{Z}^+$ and $c : E(G) \to \{\mathit{red}, \mathit{green}\}$, and a budget k.

Question: Is (G, f, k) a YES-instance of EDITUBD where the c colors set of edges that satisfy degree constraints with minimum number of new components A^* red, and all the edges of the solution skeleton of A^* green?

Note that for all reductions that only involve edge deletions that also reduce k by the appropriate amount the reverse direction of this implication is trivial, as we only use editing operations permitted by the budget and problem to alter the graph. For any recoloring of the edges (changes to c) we have to show that given a correct coloring the new coloring is also correct.

This algorithm finds the set of edges to delete that satisfies all vertices of X, while disconnecting the minimum number of components with this coloring. If we end up with more than one component and $k \geq 0$ after satisfying all vertices we use the general connection Lemma to give the final answer to our instance.

The first steps taken in reobtaining the solution is to use the information that c provides if it is a correct coloring of a YES-instance, by deleting any edges that add extra connectivity between vertices of X.

Reduction Rule 1 (Connectivity-adding edges)**.** Let G_{green} be the edge-induced subgraph of G induced on all edges colored green, and $\mathcal{C}_{\mathit{green}}$ be the set of connected components of G_{green}.

If there exists a red edge e from an unsatisfied vertex v in one component $C_i \in \mathcal{C}_{\mathit{green}}$ to another unsatisfied vertex u in a component $C_j \in \mathcal{C}_{\mathit{green}}$ with $i \neq j$, delete the edge e and reduce k by 1.

If there exists a vertex w of $G - X$ that has non-self-loop edges to two or more different components of $\mathcal{C}_{green}$, delete all red edges going to different components than the component of w in $\mathcal{C}_{green}$ and reduce k by the number of edges deleted.

Lemma 3.4.4. *The instance (G', f, k', c) after applying rule 1 is a* YES*-instance if and only if (G, f, k, c) is a* YES*-instance.*

Proof. For the forward direction of this proof we assume that (G, f, k, c) is a YES-instance, i.e. c is a correct coloring of the solution skeleton and the graph can be made connected while satisfying all degree constraints in k or less edge edits. As c is a correct coloring every vertex of X will be in the connected component $\mathcal{C}_{green}$ that it is in before deleting any edges. The edge e merges two connected components of $\mathcal{C}_{green}$, so if c is correctly colored then e must be deleted.

If we have a vertex of $G - X$ such that it connects different components of $\mathcal{C}_{green}$ with red edges, all these red connection edges must be deleted if this is a correct coloring. Finally, the solution skeleton requires that any vertex of $G - X$ which has some but not all non self-loop edges incident deleted in the solution must have at least one edge colored green – if no edges are colored green and we have to delete an edge incident to this vertex then the fact that this is a correct coloring requires that we delete all non-self-loops.

The reverse direction of this proof is trivial, as we only do legal edits permitted by the problem definitions using our budget. We also don't change the coloring c, so if c is a correct coloring in our reduced instance, it must also be a correct coloring in our original instance.

□

Claim 5. *Reduction rule 1 can be checked and implemented in time $\mathcal{O}(|V| \cdot |E|)$.*

Proof. We first obtain the graph G_{green} in time $\mathcal{O}(|V| + |E|)$. After this we run a DFS on G_{green} to identify the onnected components of G_{green}. We can identify all red edges that go between different components of G_{green} by running through the set of edges, and removing all the required edges or adjusting k is within the time bound of $\mathcal{O}(|V| \cdot |E|)$. □

Reduction Rule 2 (Edge-biconnectivity adding edges)**.** Let G_{green} be the edge-induced subgraph of G induced on all edges colored green, and G_{biconn} be the edge-biconnected components of G_{green}.

If we have a red edge e from an unsatisfied vertex u in $C_i \in G_{biconn}$ to an unsatisfied vertex v in $C_j \in G_{biconn}$, where $C_i \neq C_j$, delete e and reduce k by one.

If there exists a vertex w of $G - X$ that has non-self-loop edges to two or more different components of G_{biconn}, delete all red edges going to different components than the component of w in G_{biconn} and reduce k by the number of edges deleted.

Lemma 3.4.5. *Reduction rule 2 is safe.*

Proof. This is the same proof as the one for 1, but now we delete edges that provide edge-biconnectivity instead of connectivity over our *green* edges.

The reverse direction is true for the same reason as well, we only do edge deletions of red edges and we reduce k by the number of edge deletions. The coloring is not changed. □

Claim 6. *Reduction rule 2 can be checked and implemented in time $\mathcal{O}(|V| \cdot |E|)$.*

Proof. We first obtain the graph G_{green} in time $\mathcal{O}(|V| + |E|)$. We then find all the bridges in G_{green} with DFS [24] and remove them from G_{green}, obtaining G'_{green}, in which the connected components are G_{biconn}. As a graph has at most $|V| - 1$ bridge edges, and removing each edge in an adjacency list representation takes time proportional to the number of edges incident to a vertex, this is bounded by $\mathcal{O}(|V| \cdot |E|)$. After this we run a DFS on G'_{green} to identify the edge-biconnected components of G'_{green}. We can identify all red edges that go between different components of G_{biconn} by running through the set of edges, and removing an edge or adjusting k is within the time bound of $\mathcal{O}(|V| \cdot |E|)$. □

We can now deal with any vertex v in $G - X$ that has only red edges incident that is adjacent to two or more vertices in X. If our coloring is correct we either have to delete all edges incident to v, or keep all edges incident to v. This is because if we deleted some but not all edges incident to v in A^*, at least one of the edges of v would be colored green in a correct coloring of the solution skeleton. However we will never have to delete all edges incident to v in a correctly colored solution, as this would contradict the fact that we have colored a solution which creates the minimum number of new components.

Reduction Rule 3. If we have a vertex v in $G - X$ connected to two or more vertices in X and v and all edges incident to v' that aren't self-loops are colored red, recolor all of v's non-self-loop edges to green.

Lemma 3.4.6. *If (G, f, k, c) is* YES*-instance before recoloring the edges incident to v according to 3 the instance after recoloring the edges incident to v will also be a* YES*-instance.*

Proof. Assume for a contradiction that in a correctly colored solution we will delete all edges incident to v. Then we create one more component, while satisfying some degree constraints. However, we could satisfy the same number of degree constraints without deleting every edge incident to v: As v is adjacent to at least two vertices of X, x_1 and x_2 in G, we know that x_1 and x_2 have to be edge-biconnected after deleting all the edges incident to v. Otherwise we would have deleted all the edges incident during deletion step 1 or 2, as the red edges incident to v would provide additional connectivity over the green edges in the graph. Seeing as x_1 and x_2 are edge-biconnected, we could instead of deleting all the edges incident to v keep one edge between v and x_1 (or with x_2) and instead delete an edge that provides biconnectivity between x_1 and x_2. Then we would satisfy the same number of degree constraints, but we have created one fewer components to do so – which contradicts the fact that we have a correctly colored solution in which we create as few components as possible.

Because of this the edges incident to v will never be deleted in a correctly colored solution, and we can therefore safely recolor these edges to be green as they never will be deleted in a correctly colored solution. This means that if we have a correctly colored solution before recoloring the edges of v, we will also have a correctly colored solution after recoloring the edges of v.

The reverse direction of the proof is trivial, as the edges incident to v does not contribute to connectivity or edge-biconnectivity they can't be a part of the solution skeleton. As they aren't a part of the solution skeleton they can safely be recolored red instead of green, as there is no requirement for any coloring of edges that aren't part of the solution A^* or the solution skeleton. $\square$

Claim 7. *The reduction 3 can be done in time $\mathcal{O}(|V| + |E|)$.*

Proof. We can do this reduction by scanning through the edges incident to every vertex of $G - X$ that isn't a self-loop. If all these edges are colored red in c, and we have edges to at least two different vertices of X, we change the coloring of these edges to green. This takes time proportional with the degree of every vertex in $G - X$, which is bounded as a whole by $\mathcal{O}(|V| + |E|)$. $\square$

Now we want delete edges in X to satisfy degree constraints. We want to optimize the number of degree constraints each edge deletion contributes to, which in our case corresponds to a maximum f-matching of red edges internal in X.

Reduction Rule 4. Let A be the set of red edges with no endpoints in $G - X$, and let G^* be the edge-induced subgraph on A. Let B be the edges of the maximum h-matching in G^* with the function $h(v) = d(v) - f(v)$.

Output the new instance $(G' = G \triangle B, f, k' = k - |B|, c)$. This rule only applies if A is non-empty.

Lemma 3.4.7. *Reduction rule 4 is safe.*

Proof. We will first show the forward direction. Assume that while (G, f, k, c) is a YES-instance the reduced instance (G', f, k', c) is a NO-instance. As our original instance is a YES-instance we must have some set of edges A^* that when deleted satisfies all degree constraints in G while creating a minimum number of new components, such that $(G \triangle A^*, f, k - |A^*|)$ is still a YES-instance. Additionally c colors all the edges of A^* red, and the edges of the solution skeleton of A^* green. However $(G \triangle B, f, k - |B|, c)$ is not a YES-instance, which means that deleting the edges of B causes G' to be a NO-instance.

If $(G \triangle B, f, k - |B|)$ is a NO-instance then any set of edges C that satisfies degree constraints in $G \triangle B$ must cause $(G \triangle (C \cup B), f, k - |C \cup B|)$ to be a NO-instance according to Lemma 3.2.2 (the general connection Lemma). The relevant conditions that can be violated in the general connection Lemma is either the budget being too small, having too many components, or having too many sated or full components, as the relation between the total capacity in our graph and the number of components is constant in a graph with no violated degree constraints.

Firstly we can see that since the edges B are a maximum h-matching of red edges between the vertices of X, where we match each vertex in X at most the difference between the degree and the degree constraint for each vertex, any set of deletions that satisfy all degree constraints in G can't contain more deletions that lower the degree of two unsatisfied vertices than B. Therefore the number of edges deleted in this step is optimal. We also note that we will never disconnect components in this reduction rule, as that would mean that we have red edges that provide connectivity between vertices of X, which would have been deleted during previous reduction rules.

Assume that we are forced to disconnect more components later as a result deleting the set of edges B. Then there must be a red bridge edge e that is deleted later to satisfy the degree constraints of a vertex in X. The edge e can't be between two vertices of X, as that would mean that two vertices in X are only connected by a red edge, so this red edge would have been deleted in an earlier reduction rule. Therefore e must be between a vertex v of X and a vertex which has its degree constraints satisfied. We only delete edges of this kind when all the red edges incident to v are bridges. As the degree constraint for v has to be satsified in the optimal solution as well, if we are not deleting e we must still delete some bridge to satsify v. Then we would have to create one additional component, and therefore choosing these edges for our solution will never disconnect more components than necessary.

Assume that we are forced to create more sated components as a result of deleting the h-matching edges. Then we have a component C that after satisfying the degree constraints has no cycle edges and no surplus capacity. The kernelization algorithm guarantees that no sated components are in the graph $G - X$, due to reduction rule 2. Therefore this sated component must have some vertex of X in it. The only edges we delete as part of this reduction are edges between two vertices of X. For any edge we delete here, we know that this edge can't provide biconnectivity between vertices of X, as otherwise it would already be removed. Then any edge added in this step can't create any components with no cycle edges, as that would mean that we only have bridges left in the component, and we can't create any more sated components in this step than the optimal solution if we have a correctly colored YES-instance.

If any extra sated components are created due to this step, it must therefore be because deleting the h-matching edges forces us to later delete edges that cause a component to be sated, while in an optimal solution we would not have to create this extra sated component. As we only create sated components when we delete edges to satisfy degree constraints of some vertex v, the only red edges adjacent to v when we create the sated component are bridges. If the component of v is going to become a sated component, then the only red edges incident to v that are bridges must be to vertices not in X. We have only red bridges if there is no other way of satisfying the degree constraint of v, so in the optimal solution we must still disconnect a component to satisfy v. Since this is a YES-instance, there must be a way to satisfy v without creating a sated component in this case, so we are never forced to create extra sated components due to this step.

Finally, assume that we are forced to create more full components as a result of deleting the matching edges such that (G', f, k', c) is a NO-instance. As we require at most one extra edge deletion per full component in the graph (Lemma 3.2.2), the number of components we disconnect as part of satisfying all degree constraints with the edges of A^* must be the same number as the components we create as a result of the matching edges B being added. Also, the number of edge deletions used to satisfy all degree constraints in the optimal solution must be strictly less than the number of extra full components due to the edges deleted, as we can always transform a full component into a connectable component by deleting a cycle edge in the component.

Since we will create too many full components after deleting the matching edges, there must exist some vertex v in X for which deleted the matching edges causes the component of v to become full when satisfying degree constraints. This will either be because deleting the matching edges satisfies all degree constraints in the component, but it is then full, or we will have to make a sated component later when satisfying degree constraints. If the component is full immediately after removing the matching edges, there can't exist a set of edges that satisfies the degree constraints of this component with the same number of edge deletions without making the component full. This is because we never alter the connectivity of the component in this step, so we always deal with the same vertices as part of the component. As a component is full if we have no free connection spots but it does have some cycle edges, and we only delete edges between vertices of X in this step, we must use at least one more edge deletion to have free connection spots in this component in any case.

If we have a correctly colored set of edges in a YES-instance, then we can't be forced to create more full components after this than in the optimal solution. The only times we create a full component is if we delete a bridge, and the remaining component for the vertex v in X for which we are deleting an incident edge has no free capacity. Since this is forced, it means that there are no bridges to components with free capacity that we can consider, and this must also be the case for an optimal solution that disconnects as few components as possible. We never have the case that we can satisfy the same component using fewer bridge deletions, as we will prioritize deleting non-bridges first, and the only edges left that we can delete to satisfy the remaining degree constraints for any vertex in X is either to a component that is only incident to a single vertex of X, or it is an edge to a vertex that is in X before this reduction was made – and these edges can never become bridges as that would mean that they provide extra connectivity between two vertices of X and would therefore have to be colored green.

The reverse direction of the safety of this reduction rule follows from the fact that we can always obtain G' by deleting the edges of the h-matching, so if (G', f, k', c) is a YES-instance then (G, f, k, c) must also be a YES-instance as we can go to the instance (G', f, k', c) by only doing legal edits on the graph. □

Claim 8. *Reduction rule 4 can be done in time $\mathcal{O}(|V|^4 \cdot k^{2.5})$.*

Proof. We obtain the graph induced on A in time $\mathcal{O}(|V| + |E|)$. We must then do a h-matching in this graph, which has at most $2k$ vertices. The running time of h-matching depends on both the maximum number of vertices, and the maximum edge multiplicity in our graph 2.4.2. The maximum edge multiplicity in our graph is $k + 1$, which gives us a running time of the h-matching of $\mathcal{O}(|V|^4 \cdot k^{2.5})$, which dominates the running time of this reduction rule. □

Observe that after this step is done, there will be no red edges with both endpoints in X in G, as that would contradict the fact that we deleted the maximum h-matching between unsatisfied vertices with the permitted matching edges per vertex being the difference between their upper bound on degree and the lower bound on the degree. If any such edge existed in G then we could clearly extend the h-matching by this edge that has two endpoints in vertices of X. Therefore we only have edges with one vertex in X that can be deleted to satisfy the remaining vertices.

Now all the vertices of $G - X$ has either at least one green edge to a vertex in X, or they are adjacent to only one vertex of X. As we have found the optimal number of edges to delete that satisfy two degree constraints, all the remaining degree constraints must be satisfied by deleting edges that are incident to only one unsatisfied vertex. Any red edge is eligible for deletion, but we want to disconnect as few vertices as possible.

Only vertices of $G - X$ which are adjacent to only one vertex of X can be potentially disconnected, as otherwise they must have at least one green edge to some other vertex in X. The only thing we have to watch out for now is which bridges to delete if we have a choice. As we will show in the proof for the following reduction rule, the optimal way is to minimize the number of sated components we create when we have to disconnect.

Reduction Rule 5. While we have any vertex v in X we pick a red edge e incident to v to delete, in the following priority:

1. A non-bridge
2. A bridge to a vertex u which has $f(u) = d(u)$.
3. Any other bridge

If v does not have any red edges incident we answer NO.

Lemma 3.4.8. *Reduction rule 5 is safe.*

Proof. In this step we do one of three things: deleting a non-bridge edge, deleting a bridge edge, or outputting NO. We have to show that assuming this is a correctly colored YES-instance, all of these actions are safe.

First assume that we have a YES-instance and that after deleting a non-bridge e we no longer have a YES-instance. Since we satisfy the maximum number of degree constraints per deletion at this step, the fact that this causes a NO-instance must mean that after this deletion we will either disconnect more components than which are necessary in an optimal solution, or we have more sated and/or full components after the deletion as the use of the budget k is as small as possible (due to the connection Lemma 3.2.2).

If we don't disconnect any components connected to v then the resulting component with v cannot be either a sated or full component, as v is adjacent to a vertex that was satisfied before having at least one incident edge removed. Then for this to become a NO-instance it must be because of deleting a bridge edge. Assume therefore that we have a YES-instance, and that after deleting a bridge edge e we no longer have a YES-instance.

The bridge deleted can't disconnect any more components than already necessary, as all the edges incident to v must clearly be bridges, and the same amount of edges incident to v must be deleted in any solution. Then there is some other edge e' incident to v which could be deleted instead of e to maintain a YES-instance. The edge e' has to be a bridge, as otherwise we would never delete e over e'.

As we can't create more components than necessary with this rule, assume instead that we create more sated or full components than an optimal solution requires. For the component of v to be sated or full we must be totally disconnected from every component that has some free capacity or some extra cycle edges. Assume that it is possible to disconnect the same number of components, but we can have fewer sated or full components in an optimal solution. We must delete the same number of edges incident to v to satisfy the degree constraint, so if there are only bridge edges to delete then we have to disconnect some component from v. As it is possible to create fewer sated or full components in this case, we know that we have at least one component adjacent with free capacity (or otherwise we would create a full component after deletion), and this edge was deleted in our reduction but in an optimal solution we do not delete this edge. We will not delete this edge if we have bridges to components without any free capacity, so as we deleted this edge and therefore created one extra full component, there are no bridges to components with no free capacity incident to v. Then the optimal solution also has to create an extra sated component in this case: As we have to delete an edge incident to v to satisfy the degree constraint and deleting an edge to a component with free capacity will create an extra full component, the edge to the other component has to be the only edge to an other component that can be deleted. Then we have to delete this edge anyway, as the edge is the only eligible edge for deletion, and v hasn't satisfied its degree constraint.

Finally we have to show that it is safe to answer NO if we have no red edges incident to v. As we have a correctly colored instance, the edges that are part of the solution that satisfies all degree constraints must be colored red. However, since v is not satisfied and there are no red edges incident to v we must conclude that it is impossible to satisfy the degree constraints of v if the instance is a correctly colored YES-instance.

□

Claim 9. *Reduction rule 5 can be done in time $\mathcal{O}(|V| + |E|)$.*

Proof. To perform reduction rule 5 we must for a vertex in X check which edge to delete according to the rule, and delete the edge. Both identifying the vertices of X, identifying which edge to delete according to the rule, and deleting an edge can be done in time $\mathcal{O}(|V| + |E|)$. □

After exhaustively applying reduction rule 5, G is a graph with no unsatisfied vertices – if not, it was impossible to delete enough edges adjacent to a vertex in X with this coloring. If $k \geq 0$ we can then use the general connection Lemma (3.2.2) to see if this is a YES-instance or a NO-instance in time $\mathcal{O}(|V| + |E|)$.

Since every reduction rule for reobtaining the solution takes polynomial time, and either reduces the parameter k or does a recoloring of edges where the maximum number of edges to recolor is clearly bounded by $|E|$, the algorithm as a whole takes polynomial time.

3.4.4 Randomized algorithm

Now we have all the parts necessary for a randomized algorithm for our problem.

Theorem 3.4.9 (Randomized algorithm for EDITUBD)**.** EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES *has a randomized algorithm that runs in polynomial time and outputs* YES *if this is a* YES*-instance with probability* $\geq \frac{1}{2^{12k}}$*, and* NO *otherwise.*

Proof. First we run the kernelization algorithm on our instance, which runs in polynomial time. Then we randomly color all edges independent of each other either green or red, with probability $\frac{1}{2}$. After this we try to delete the necessary edges to satisfy degree constraints, given that this coloring is correct. Reobtaining the solution takes polynomial time. If this instance is a YES-instance we will color the necessary $< 12k$ edges of the solution edges A^* and its corresponding solution skeleton to reobtain the solution with probability at least $\frac{1}{2^{12k}}$. Reobtaining the solution works if we have a correct coloring, so this gives us a randomized algorithm that runs in polynomial time with success probability at least $\frac{1}{2^{12k}}$. □

Success probability here is defined as the chance that if this particular instance is a YES-instance, then the algorithm will also output YES. Note that for any NO-instance, the algorithm will always answer NO. If we repeat this algorithm 2^{12k} times, we have a probability of answering YES given that this is a YES-instance of at least $\frac{1}{e}$ [5, Chapter 5]:

$$(1 - \frac{1}{2^{12k}})^{2^{12k}} \leq (e^{-\frac{1}{2^{12k}}})^{2^{12k}} = \frac{1}{e}$$

We therefore have a randomized algorith for EDITUBD which has a success probability $\geq \frac{1}{e}$ independent of k with a running time of $2^{12k} \cdot n^{\mathcal{O}(1)}$.

Derandomization

Even if we can obtain any probability bound we want for our algorithm with enough repetitions, we still can't give a 100% certainity that any NO-answer is correct with this method. However, if we are able to go through all the different possible ways to color a solution skeleton, we guarantee that we will hit the correct coloring at least once if it exists. The Theorem for the construction of the universal set and also the definition of an universal set is both by Cygan et. al. [5, Chapter 5.6].

Definition 3.4.1 ((n, k)-universal set)**.** An (n, k)-*universal set* is a family $\mathcal{U}$ of subsets of $[n]$ such that for any $S \subseteq [n]$ of size k, the family $\{A \cap S : A \in \mathcal{U}\}$ contains all 2^k subsets of S.

If we take n to be the number of edges we have in our instance after kernelization, and take k to be the maximum size of our solution A^* plus the edges of the corresponding solution skeleton, then if we iterate through the separations of the $(\mathcal{O}(k^4), 12k)$-*universal set*, then one of these sets has to correctly separate the edges of the solution A^* and the solution skeleton if this is a YES-instance. The way we obtain a coloring from one of the sets in the universal set is by coloring every edge in the set one color, and every edge not in the set the other color. As we hit every subset we will necessarily obtain a correct coloring of the needed edges. From the same chapter in [5] we have a Theorem that states that we can construct this universal of limited size in limited time:

Theorem 3.4.10. *For any $n, k \geq 1$ one can construct an (n, k)-universal set of size $2^k k^{\mathcal{O}(\log k)} \log n$ in time $2^k k^{\mathcal{O}(\log k)} n \log n$.*

This Theorem allows us to prove the following result:

Theorem 3.4.11. EDGE EDITING TO A CONNECTED GRAPH OF UPPER BOUNDED DEGREES *has a deterministic algorithm that runs in time $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$.*

Proof. We apply Theorem 3.4.10 and obtain an $(16k^4, 12k)$-*universal set* of size

$$2^{12k}(12k)^{\mathcal{O}(\log 12k)} \log 16k^4$$

in time

$$2^{12k}(12k)^{\mathcal{O}(\log 12k)} 16k^4 \log 16k^4$$

If we go through each of the

$$2^{12k}(12k)^{\mathcal{O}(\log 12k)} \log 16k^4$$

colorings defined by the universal set and try to reobtain the solution in polynomial time, then we have a deterministic algorithm for our problem with running time of

$$2^{12k}(12k)^{\mathcal{O}(\log 12k)}16k^4 \log 16k^4 + 2^{12k}(12k)^{\mathcal{O}(\log 12k)} \log 16k^4 \cdot n^{\mathcal{O}(1)}$$

$$\in 2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$$

. □

The proof is slightly imprecise, as the bound on the number of edges after kernelization isn't exactly $16k^4$, but this does not change the final asymptotic bound on the runtime of the algorithm.

3.5 NP-Completeness

As we now have demonstrated the existence of a polynomial kernel and a single exponential FPT algorithm we will round off the chapter on EDITUBD by proving that is NP-complete, and giving a lower bound for the running time assuming ETH.

Theorem 3.5.1. EDITUBD *is NP-complete, and assuming ETH holds no algorithm with running time* $2^{o(k)} \cdot n^{\mathcal{O}(1)}$ *exists.*

Proof. To show that a problem P is NP-complete, it is sufficient to show that it is both NP-hard and in NP. First, we show that it is NP-hard by reducing an arbitrary instance of a known NP-complete problem to EDITUBD in polynomial time. The problem we reduce from is HAMILTONIAN CYCLE, which is NP-complete [11].

HAMILTONIAN CYCLE
Input: An undirected simple graph $G = (V, E)$
Question: Does a cycle C exist in G such that $V(C) = V(G)$?

As this is a parameterized reduction where the parameter k is linear in size with regards to the number of vertices and edges in the graph, we also obtain the ETH lower bound from this reduction. This is because HAMILTONIAN CYCLE has a linear reduction from 3-SAT, meaning that any $2^{o(n+m)}$ algorithm would violate ETH [5, Theorem 14.6].

EDITUBD is NP-hard

Claim: G is a YES-instance of HAMILTONIAN CYCLE if and only if the same graph with the degree constraint $f(v) = 2$ for every vertex in G, and budget $k = |E(G)| - |V(G)|$ is a YES-instance of EDITUBD.

Forward direction: HAMILTONIAN CYCLE with G is a YES-instance, so there exists a cycle C in G with $V(C) = V(G)$. Since there are $|V(G)|$ vertices in the cycle, it also has $|V(G)|$ edges. The number of edges not in the cycle is $|E(G)| - |V(G)|$. Let $G_C = (V(G), E(C))$ be the graph G with all edges other than the ones in the cycle removed. Every vertex in G_C has degree 2, and G_C is a connected graph. Since G_C can be obtained from G by removing $|E(G)|-|V(G)| = k$ edges, and it satisfies the constraint on degrees and connectivity, we have a YES-instance of EDITUBD.

Backward direction: If EDITUBD is a YES-instance, we have constructed a graph G' with $\leq k$ edge additions and deletions, where G' is connected and every vertex in G' has degree of at most 2. A connected graph where every vertex has degree 2 has a cycle [25, Lemma 6.1], which must necessarily be the entire graph as the graph is connected. If every vertex in G' has degree 2, then G' must be a cycle, and if all edges in G' exist in G then a Hamiltonian cycle exists in G as well.

In any graph, $\sum_{v \in V} d(v) = 2|E|$ [25, Handshaking Lemma]. All vertices in G' has degree 2. Assume otherwise, that there exists a vertex v in G' with $d(v) < 2$. Since G' is connected, v must have degree 1. With the sum of degrees equaling $2|E|$ and one vertex having degree 1, we must have at least one other vertex with odd degree, as any even number plus an odd number must be odd. All vertices in G' has degree of at most 2, so we have at least two vertices that have degree 1.

$$|E(G')| = \frac{1}{2} \sum_{v \in V(G')} d(v) \leq \frac{2|V(G')| - 2}{2} = |V(G')| - 1$$

To have $\leq |G'| - 1$ edges in G', we must have removed $\geq |E(G)| - (|V(G')| - 1) = |E(G)| - |V(G)| + 1$ edges from G. However, this contradicts the maximum number of edits we can do, which is $k = |E(G)| - |V(G)| < |E(G)| - |V(G)| + 1$, with our only assumption being the existence of a vertex in G' with degree less than 2.

G' is a connected graph with all vertices having degree 2, so it must be a cycle with all the nodes in the graph. As the vertices of G and G' are the same, we only need to show that all the edges of the cycle exist in G.

$$2|E(G')| = \sum_{v \in V(G')} d(v) = \sum_{v \in V(G')} 2 = 2|V(G')|$$

Since G and G' has the same vertices, $|V(G)| = |V(G')|$. The difference in number of edges between G and G' is

$$|E(G)| - |E(G')| = \frac{2|E(G)| - 2|V(G)|}{2} = |E(G)| - |V(G)| = k$$

As G' has k less edges than G, and the total number of edits $\leq k$, all the operations we did must be edge deletions. Then as G' is a cycle with all the vertices of G, and all the edges in G' exists in G, HAMILTONIAN CYCLE is a YES-instance. Therefore EDITUBD is NP-hard, and as we have a linear reduction from HAMILTONIAN CYCLE any $2^{o(k)} \cdot n^{\mathcal{O}(1)}$ algorithm for EDITUBD would contradict ETH.

EDITUBD is in NP

Now, for EDITUBD to be NP-complete, all that remains is showing that it is in NP.

Given a solution A to an instance (G, k, f) of EDITUBD, we must verify in polynomial time that it is a valid solution.

Verify that $|A| \leq k$. Construct G' by going through the edges in A and either add the edge if it does not exist in G, or deleting the edge if it exists in G. Now, scan through G', and for each vertex v verify that $d(v) \leq f(v)$. If this holds for every vertex, we finally check that the entire graph is connected. If this is true then we have a valid solution, and if it is not the solution is invalid.

The verifier runs in polynomial time, and as EDITUBD is both in NP and NP-hard, it is NP-complete. □

Any algorithm with a running time of $2^{o(k)} \cdot n^{\mathcal{O}(1)}$ would violate ETH, so assuming ETH holds the algorithm from section 3.4 with a running time of $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$ cannot be significantly improved. Of course, this is slightly misleading due to the constant factor hidden in the exponent with the big $\mathcal{O}$ notation. Nevertheless this shows an asymptotically tight bound for EDITUBD assuming ETH.

Chapter 4

Lower Bounded Degrees

4.1 Problem statement

We already introduced this problem in the introduction, but we state it again here for convenience.

EDGE EDITING TO A CONNECTED GRAPH OF LOWER BOUNDED DEGREES

Input: A graph $G = (V, E)$, an integer k, a function $f : V(G) \to \mathbb{Z}^+$

Question: Does a set $A \subseteq \binom{V(G)}{2}$ exist such that $G' = (V(G), E(G) \triangle A)$ is a connected graph, all vertices v in $V(G')$ has $d(v) \geq f(v)$, and $|A| \leq k$?

EDGE EDITING TO A CONNECTED GRAPH OF LOWER BOUNDED DEGREES, or EDITLBD for short, is similar to EDITUBD, but now we have a lower bound and not an upper bound. It is easy to see that we don't have to consider any edge deletions in this version of the problem, as any solution with an edge deletion will also be a solution without that edge deletion.

Due to the way the minimum degree requirement and the connectivity requirement are both inclined toward edge addition, we can hope that this problem is easier than the upper-bounded version. Indeed this is the case, and we will give a polynomial time algorithm for EDITLBD.

First we will for convenience define what we mean by an unsatisfied vertex and the set X:

Definition 4.1.1 (Unsatisfied vertex and X)**.** We say that a vertex v in G is *unsatisfied* if $d(v) < f(v)$. The set $X \subseteq V(G) = \{v \mid d(v) < f(v)\}$ is the set of all unsatisfied vertices. Any vertex not in X is a *satisfied* vertex.

In this section we will assume that for every vertex in the graph $f(v) < |V|$ holds, as the maximum degree of a vertex in a simple graph is $|V| - 1$, which happens when it is adjacent to every other vertex in the graph.

4.2 Polynomial algorithm

The polynomial algorithm is constructed as a series of reduction rules, which with exhaustive application will give the answer to the given instance of EDITLBD. All the reduction rules take polynomial time, and will be executed a polynomial number of times, resulting in a polynomial algorithm.

The first reduction we will do is alter the lower bound function to ensure that every connected component has at least one unsatisfied vertex, which removes some edge cases from later consideration.

Reduction Rule 1. If we more than one connected component in our graph and a connected component C in our graph which has no unsatisfied vertices, pick an arbitrary vertex v in C and set $f'(v) = d(v) + 1$.

Lemma 4.2.1. *Reduction rule 1 is safe.*

Proof. For the forward direction we assume that (G, f, k) is a YES-instance. Then there exists a set of edges A such that $G \triangle A$ is a connected graph with all degree constraints f satisfied, and $|A| \leq k$. As C is connected to the rest of the graph, we must have an edge $\{u, c\}$ in A where $c \in C$ and $u \notin C$. Then $A' = A - \{u, c\} + \{u, v\}$ must also be a solution, as we still connect the component C to the rest of the graph, and as c already has enough edges to satisfy its degree constraints c must also be satisifed in $G - A'$. As A' adds one edge incident to v, A' is also a solution for (G, f', k) as all degree constraints are satisfied and the entire graph is connected.

For the reverse direction we assume that (G, f', k) is a YES-instance, so we have a set of edges A such that $G \triangle A$ is a connected graph and all degree constraints f' are satisfied. Then the same set A will satisfy the instance (G, f, k) as the degree constraints are the same for every vertex except v, where it is less strict. $\square$

After exhaustive application of reduction rule 4.2.1 every connected component in the graph has at least one unsatisfied vertex.

The solution for the problem will gradually be built as a set of edges to add to the graph S which is initally empty, constructed using a series of reduction rules. The safety of the reduction rules depend on the fact that $|S|$ is at any time the $\leq$ *least* amount of edges that must be added to connect the entire graph and satisfy all degree constraints. For $S = \emptyset$ this is obviously true.

The reduction rules are applied in the listed order, if the conditions of the reduction rule applies.

Reduction Rule 2. If there exists some unsatisfied vertices in $G + S$, let $G' = \overline{G[X]}$ and $g : V(G') \to \mathbb{Z}^+$, $g(v) = f(v) - d_G(v))$. Find the edges M of the maximum g-matching in G', and set $S = M$. While there exists any unsatisfied vertex v in $G + S$ add any edge $\{v, c\} \in E(G - S)$ to S, where c is any vertex of G.

Lemma 4.2.2. *After applying reduction rule 2, $|S|$ is equal to or smaller than the minimum number of edges that needs to be added to satisfy all degree constraints in G and connect the graph.*

Proof. When satisfying degree constraints, the most we can do in one operation is to add an edge between two vertices that does not satisfy the degree constraint. If we find the maximum number of edges that satisfies two degree constraints, we know that this gives us a lower bound on the number of edges needed to solve this problem. The possible edges that satisifies two degree constraints are the edges between two unsatisfied vertices in G that does not exist in G. However, we can't add more than $f(v) - d(v)$ edges to an unsatisfied vertex v without satisfying it. Finding the maximum number of edges that can satisfy two degree constraints in this way corresponds exactly to the g-matching problem in G' with the function value g. Thus the size of S must be fine after the edges of the g-matching are added.

Then for the reduction rule to not be safe we must add too many edges after the g-matching edges are added. Assume that this is the case, ie. that we add a set of edges after the g-matching ending up with S, but there exists some set B such that all vertices in $G - B$ are satisfied, and that the size of B is less than S. As every edge in S satisfies at least one degree constraint, for B to be smaller then it has to have more edges that satisfies two degree constraints. However this can't be true, as the number of edges that satisfy two degree constraints in S corresponds to a maximum g-matching, so B can only be smaller if the matching is not a maximum g-matching. □

As all degree constraints are now satisfied, the remaining reduction rules will connect as many components as possible without increasing the size of S needlessly. To do this we will remap edges in S to become bridges while satisfying the same number of vertices whenever possible, as then we both satisfy degree constraints and merge components at the same time.

Reduction Rule 3 (Solution is too big)**.** If $|S| > k$, output NO.

Lemma 4.2.3. *Reduction rule 3 is safe.*

Proof. As $|S|$ is smaller or equal to the minimum number of edges required to be added to solve this instance, we can't solve this instance if we must use more than k edits. □

Reduction Rule 4 (Solution found)**.** If $G + S$ is a connected graph, and all degree constraints are satisfied, output YES.

Lemma 4.2.4. *Reduction rule 4 is safe.*

Proof. As $|S| \leq k$ and $G + S$ satisfies both the degree constraints and connects the graph, S is a solution of our original problem and we can just output YES. □

Reduction Rule 5 (Matching reordering)**.** If we have edges $\{u, v\}$ and $\{a, b\}$ in S, where u and a are in different connected components in $G + S$, and either $\{u, v\}$ or $\{a, b\}$ is not a bridge in $G + S$, let $S' = (S \setminus \{\{u, v\}, \{a, b\}\}) \cup \{\{u, a\}, \{v, b\}\}$.

Lemma 4.2.5. *Reduction rule 5 is safe.*

Proof. Since $|S| = |S'|$ and we do not have any unsatisfied vertices in $G + S'$, this reduction does not violate the fact that there can't exist any smaller set than S that fixes all degree constraints and also connects the graph. □

After exhaustively applying the previous reduction rules we have no unsatisfied vertices, and all of the edges in S are bridges. The initial guarantee on the optimal size of S comes from the fact that there does not exist a smaller set of edge additions that satisfies all degree constraints. We also know that the optimal way of connecting a set of connected components is by only adding bridges, reducing the number of connected components by one for each bridge added. As we already have only bridges added, if we still have more than one connected component we must add more bridges between components to the graph.

Reduction Rule 6 (Connect remaining components)**.** If we have more than one connected component in $G+S$ pick two vertices c_1, c_2 from different connected components in $G+S$ and add the edge $\{c_1, c_2\}$ to S.

Lemma 4.2.6. *Reduction rule 6 is safe.*

Proof. As every edge in S is a bridge, we can't connect more components with the same or smaller number of edge additions. Since the edge $\{c_1, c_2\}$ must also be a bridge (as otherwise c_1 and c_2 must be part of the same connected component), we still use the optimal number of edge additions needed to satisfy all degree constraints and connect the entire graph. □

At any time at least one of the reduction rules listed will apply, as we have covered the cases where we actually have a solution, when the solution is too big, and the final case when we haven't connected all components in $G + S$. Thus the reduction rules provided gives us a deterministic algorithm for any instance of EDITLBD.

4.3 Analysis and results

The running time of our algorithm depends on the time needed for each step, and the number of steps taken.

Theorem 4.3.1 (EDITLBD algorithm)**.** *Given an instance (G, f, k) of* EDITLBD *we can in time $\mathcal{O}(|V(G)|^4)$ answer* YES *if (G, f, k) is a* YES*-instance, or* NO *if (G, f, k) is a* NO*-instance.*

Proof. The first reduction rule where we make sure that every connected component has at least one unsatisfied vertex can be done in linear time by searching through each connected component and setting one vertex to be unsatisfied if none of the vertices of a given component are unsatisfied. After this we do a f-matching in an auxilliary graph with a subset of our vertices, which runs in time $\mathcal{O}(|V|^4)$ as this is a simple graph (Theorem 2.4.2).

After the first two reduction rules, the remaining steps are done by reduction rules 3, 4, 5 and 6. These reduction rules can be both checked and done in time bounded by $\mathcal{O}(|V| + |E|)$ (linear time): Reduction rule 3 only identifying if the size of S is bigger than our budget. The remaining reduction rules requires constructing the graph $G + S$, which can be done in linear time. Seeing if all degree constraints are satisfied in $G+S$ can be done by running a graph search through the graph, which takes linear time. Identifying the bridges in the graph can be done with DFS [24], which at the same time can identify the connected components of our graph. Altering the set S can also be done in linear time.

The reduction rules that modify S decreases the number of components in $G + S$ by one every time the reduction rule applies. As there are at most $|V|$ components in our graph these reduction rules can apply at most $|V|$ times. Then the running time of the reduction rules except for the f-matching is bounded by $\mathcal{O}(|V|(|V| + |E|))$. Combining both results the running time of the algorithm is bounded by $\mathcal{O}(|V| \cdot (|V| + |E|) + |V|^4) = \mathcal{O}(|V|^4)$. The dominating part of the running time of our algorithm is the f-matching in the auxilliary graph. □

Chapter 5

Conclusion

5.1 Summary

We have shown that EDITUBD is FPT when parameterized by the number of edits k, and both given a polynomial kernel with $\mathcal{O}(k^4)$ edges and $\mathcal{O}(k^3)$ vertices, and an algorithm based on the technique of random separation with the running time $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$, which is asymptotically optimal assuming that ETH holds. For EDITLBD we have given a polynomial time algorithm based on matching techniques.

5.2 Open problems and future work

Since we know that deleting the correct edges to satisfy all the degree constraints is the hard part of EDITUBD, can the kernel be asymptotically improved especially regarding the number of edges?

An interesting extension of the two problems considered in this thesis is the combination of both a lower and an upper bound on the degree constraint:

EDGE EDITING TO CONNECTED GRAPH WITH INTERVAL BOUNDED DEGREES

Input: A graph $G = (V, E)$, an integer k, two functions $f : V(G) \to \mathbb{Z}^+$ and $g : V(G) \to \mathbb{Z}^+$

Question: Does a set $A \subseteq \binom{V(G)}{2}$ exist, such that $G' = (V, E'), E' = E \triangle A$ is a connected graph, $\forall v \in V(G') : g(v) \leq d_{G'}(v) \leq f(v)$, and $|A| \leq k$?

As this problem can encode the problem of EDGE EDITING TO A CONNECTED GRAPH OF GIVEN DEGREES by setting $f(v) = g(v)$ for all vertices this problem must be at least as hard as the connected problem with exact degree constraints.

A natural extension to EDITLBD and EDITUBD is to allow for the additional editing operation of *vertex deletion* or potentially *edge contraction*.

In the same way that considering bounded degrees is a variation on EDGE EDITING TO A CONNECTED GRAPH OF GIVEN DEGREES, we can consider a bounded version of EDGE EDITING TO A GRAPH OF GIVEN DEGREE SEQUENCE [14], where instead of achieving the exact degree sequence specified we instead ask if we can edit to a connected graph with a degree sequence that is bounded by the given degree sequence. Similar to the problem for a given degree sequence neither the lower bounded or upper bounded version of this problem is FPT parameterized by k only, but additional parameterization such as maximum degree is unknown.

Bibliography

[1] Burzyn, P., Bonomo, F., and Durán, G. (2006). NP-completeness results for edge modification problems. *Discrete Applied Mathematics*, 154(13):1824–1844.

[2] Cai, L. (1996). Fixed-parameter tractability of graph modification problems for hereditary properties. *Inf. Process. Lett.*, 58(4):171–176.

[3] Cai, L., Chan, S. M., and Chan, S. O. (2006). Random separation: A new method for solving fixed-cardinality optimization problems. In Bodlaender, H. L. and Langston, M. A., editors, *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006, Zürich, Switzerland, September 13-15, 2006, Proceedings*, volume 4169 of *Lecture Notes in Computer Science*, pages 239–250. Springer.

[4] Chen, J., Kanj, I. A., and Xia, G. (2006). Improved parameterized upper bounds for vertex cover. In Kralovic, R. and Urzyczyn, P., editors, *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, volume 4162 of *Lecture Notes in Computer Science*, pages 238–249. Springer.

[5] Cygan, M., Fomin, F. V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., and Saurabh, S. (2015). *Parameterized Algorithms*. Springer.

[6] Diestel, R. (2012). *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer.

[7] Downey, R. G. and Fellows, M. R. (1995). Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theor. Comput. Sci.*, 141(1&2):109–131.

[8] Downey, R. G. and Fellows, M. R. (1999). *Parameterized Complexity*. Monographs in Computer Science. Springer.

[9] Fomin, F. V., Golovach, P. A., Panolan, F., and Saurabh, S. (2016). Editing to connected f-degree graph. In Ollinger, N. and Vollmer, H., editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 36:1–36:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[10] Fomin, F. V., Lokshtanov, D., Saurabh, S., and Zehavi, M. (2019). *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press.

[11] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[12] Golovach, P. A. (2015). Editing to a graph of given degrees. *Theor. Comput. Sci.*, 591:72–84.

[13] Golovach, P. A. (2017). Editing to a connected graph of given degrees. *Inf. Comput.*, 256:131–147.

[14] Golovach, P. A. and Mertzios, G. B. (2017). Graph editing to a given degree sequence. *Theor. Comput. Sci.*, 665:1–12.

[15] Impagliazzo, R. and Paturi, R. (2001). On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375.

[16] Khot, S. and Raman, V. (2002). Parameterized complexity of finding subgraphs with hereditary properties. *Theor. Comput. Sci.*, 289(2):997–1008.

[17] Lewis, J. M. and Yannakakis, M. (1980). The node-deletion problem for hereditary properties is NP-complete. *J. Comput. Syst. Sci.*, 20(2):219–230.

[18] Lovász, L. and Plummer, M. D. (2009). *Matching theory*, volume 367. American Mathematical Soc.

[19] Mathieson, L. and Szeider, S. (2012). Editing graphs to satisfy degree constraints: A parameterized approach. *J. Comput. Syst. Sci.*, 78(1):179–191.

[20] Micali, S. and Vazirani, V. V. (1980). An $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 17–27. IEEE Computer Society.

[21] Moser, H. and Thilikos, D. M. (2006). Parameterized complexity of finding regular induced subgraphs. In Broersma, H., Dantchev, S. S., Johnson, M., and Szeider, S., editors, *Algorithms and Complexity in Durham 2006 - Proceedings of the Second ACiD Workshop, 18-20 September 2006, Durham, UK*, volume 7 of *Texts in Algorithmics*, pages 107–118. King's College, London.

[22] Natanzon, A., Shamir, R., and Sharan, R. (2001). Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113(1):109–128.

[23] Sipser, M. (1997). *Introduction to the theory of computation*. PWS Publishing Company.

[24] Tarjan, R. E. (1974). A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2(6):160–161.

[25] Wilson, R. J. (1979). *Introduction to graph theory*. Pearson Education India.