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

On Greedy-Branching for Vertex Deletion to Tree-Like Graphs

Bachelor thesis

von Vincent Borko

zur Erlangung des Grades „Bachelor of Science“ (B. Sc.)
im Studiengang Computer Science (Informatik)

Erstgutachter:	Prof. Dr. Rolf Niedermeier
Zweitgutachter:	Prof. Dr. Markus Brill
Betreuer:	Hendrik Molter, Prof. Dr. Rolf Niedermeier

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, den _____ _____

Datum Unterschrift

Zusammenfassung

In dieser Arbeit betrachten wir den Graphparameter *Arboricity* und stellen einen Algorithmus vor, welcher die Arboricity eines Graphen durch Löschen von Knoten reduziert. Die Arboricity $\text{arb}(G)$ eines Graphen G ist definiert als die minimale Anzahl von Wäldern, welche durch das Partitionieren der Kanten von G induziert werden kann. Wir definieren das Problem Arboricity-two-reduction, welches fragt ob wir die Arboricity eines gegebenen Graphen G auf höchstens zwei reduzieren können indem wir maximal k Knoten löschen. Wir entwickeln einen Algorithmus für dieses Problem und analysieren dessen Laufzeit. Unser Arboricity-two-reduction Algorithmus ist inspiriert von einem Algorithmus von Cao [1], welcher das Problem Feedback Vertex Set löst. Feedback Vertex Set hat große Ähnlichkeit zu Arboricity-two-reduction, da Feedback Vertex Set lediglich eine andere Bezeichnung für Arboricity-one-reduction ist. Ein Graph hat Arboricity eins genau dann wenn er ein Wald ist. Caos Algorithmus verwendet eine Greedy-Branching Methode, welche immer den Knoten mit maximalem Knotengrad betrachtet und diesen entweder zur Lösungsmenge hinzufügt, oder in eine Knotenmenge aus nicht löschbaren Knoten hinzufügt, falls der Algorithmus keine Lösung mit diesem Knoten in der Lösungsmenge findet. Unser präsentierter Algorithmus für Arboricity-two-reduction verwendet denselben Ansatz mit zwei Hauptmodifikationen. Wir brauchen einerseits neue Abbruchbedingungen für die Rekursion und einen neuen Rekursionsbasisfall. Unser Basisfall ist in der Lage Arboricity-two-reduction für Graphen mit maximalem Knotengrad vier zu lösen. Wir zeigen, dass unser Algorithmus für Arboricity-two-reduction in FPT-Zeit läuft. Für die Zukunft wollen wir den allgemeinen Fall arboricity-p-reduction für ein beliebiges $p \in \mathbb{N}$ lösen können. Unser Algorithmus dient als Ausgangspunkt für die weitere Erforschung, um Muster zu finden, welche den Algorithmus verallgemeinern können. Arboricity-Zero-Reduction ist ein anderer Begriff für Vertex Cover. Daher gibt es jetzt, einschließlich unserer Arbeit, Algorithmen zum Lösen von Arboricity-zero-reduction, Arboricity-one-reduction und Arboricity-two-reduction, von welchen wir den Algorithmus erweitern können. Unseres Wissens nach sind wir die Ersten, die die algorithmische Komplexität von Arboricity-two-reduction analysieren und einen Algorithmus liefern.

Abstract

In our work we consider the graph parameter *arboricity* and deliver an algorithm that reduces the arboricity by deleting vertices from the graph. The arboricity arb(G) for a graph G is defined as the minimum numbers of forests into which its edges can be partitioned. We define the specific problem Arboricity-two-reduction where we are asked to reduce the arboricity of a given graph G to two by deleting at most k vertices. We develop and algorithm for this problem and analyse its running time. Our Arboricity-two-reduction algorithm is inspired by an algorithm due to Cao [1], which solves the problem Feedback Vertex Set. The Feedback Vertex Set is strongly related to Arboricity-two-reduction, since Feedback Vertex Set is just a different term for Arboricity-one-reduction. Arboricity one is equivalent to the graph being a forest. Cao's algorithm uses a greedy-branching method, which always considers the maximum degree vertex and stores it into either the solution set or, if no solution is found containing this vertex, into an undeletable vertex set out of which the solution cannot contain vertices. Our proposed algorithm for Arboricity-two-reduction uses the same approach with two major modifications. We need different return conditions for the recursion and a new recursion base case. Our base case will be able to solve Arboricity-two-reduction for maximum-degree-four graphs. We show that our algorithm for Arboricity-two-reduction runs in FPT-time, parameterized by k. For the future we desire to solve the general case of arboricity-p-reduction for any $p \in \mathbb{N}$. Our algorithm serves as a starting point for further exploration to find patterns to be able to generalize the algorithm. Arboricity-zero-reduction is a different term for Vertex cover. Hence, including our work there now exist algorithms to solve Arboricity-zero-reduction, Arboricity-one-reduction, and Arboricity-two-reduction, from which we can extend in the future. To the best of our knowledge we are the first to analyse the computational complexity of Arboricity-two-reduction and provide an algorithm.

Contents

1 Introduction **11**
- 1.1 Related work . . . 13
- 1.2 Our contribution . . . 13
- 1.3 Organization of the work . . . 13

2 Preliminaries **15**

3 Feedback Vertex Set **17**

4 Arboricity-Two-Reduction **23**
- 4.1 Algorithm . . . 23
- 4.2 Description of Algorithm 2 . . . 24

5 Recursion Base Case **29**
- 5.1 Arboricity-Two-Reduction for Maximum Degree Four . . . 29
- 5.2 Processing vertices connected to $G[Q]$ . . . 35

6 Analysis of Algorithm 2 **41**
- 6.1 Correctness . . . 41
- 6.2 Running Time . . . 44

7 Conclusion **47**
- 7.1 Future Work . . . 47

Literature **49**

1 Introduction

In theoretical computer science a big part of algorithmics and computational complexity analysis deals with solving graph problems. Graphs allow us to abstract complex problems and to elegantly reflect them. Many of those problems do not just appear in a theoretical environment but also in real life. Graph modifications are used to transform a graph into a desired state. A graph is a combination of vertices and edges, connecting the vertices. Modifying a graph means changing anything that affects either the vertices or the edges of a graph, for example adding or removing vertices or edges. Complex problems in graph theory often become easier to solve on tree-like graphs. So the question arises how we can measure tree affinity. In our paper we look at the arboricity of a given undirected graph and figure out how to decrease it by deleting vertices. Intuitively, the lower the arboricity of a graph, the more resemblence it has to a tree. The arboricity $\text{arb}(G)$ is defined as the minimum numbers of forests into which its edges can be partitioned. A forest F is a graph without cycles. A graph contains a cylce, if there a path of edges and vertices from which a vertex of the graph is reachable from itself. For our specific problem we reduce the arboricity by deleting vertices from the graph. We define ARBORICITY-p-REDUCTION as the problem of finding at most k vertices of a given graph G, such that if those vertices are removed from G, then the resulting graph has arboricity at most p. The output of the algorithm either returns the set X of deleted vertices or "NO" if no such vertex set of size at most k exists.

ARBORICITY-p-REDUCTION
Input: An undirected graph G and an integer $k \in \mathbb{N}$.
Question: Can the graph be reduced to arboricity p by deleting at most k vertices?
Output: A vertex deletion set $X \subseteq V(G)$ of size at most k or "NO".

Let K_n be the complete graph with n vertices. In the given example (see Figure 1.1) every vertex has the same degree as the graph is complete. The graph has arboricity three, each of the three forests is marked by a different edge layout. One can easily observe that deleting any vertex will have the same effect on the arboricity. Whichever vertex you delete will reduce the arboricity to two. Likewise deleting any combination of three vertices reduces the arboricity to one.

We will show in our work that ARBORICITY-TWO-REDUCTION is contained in FPT with parameter k. It is easy to see that having bounded arboricity is hereditary. Lewis and Yannakakis [12] showed that vertex deletion to hereditary graph classes is NP-hard. Hence, ARBORICITY-TWO-REDUCTION is an NP-hard problem. A related but different,

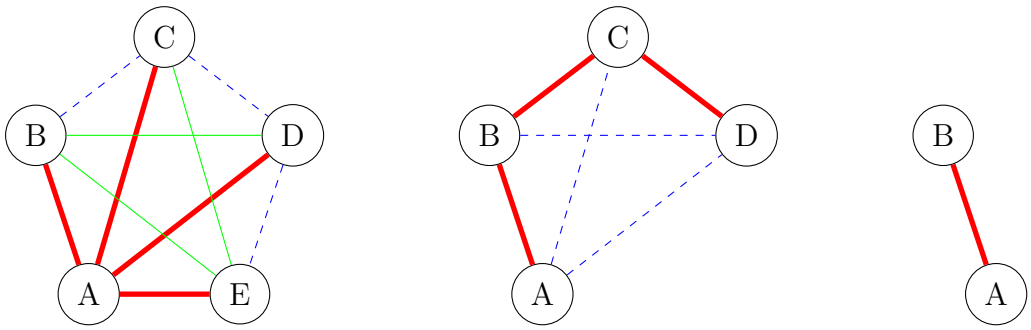

Figure 1.1: A K_5-graph with arboricity three. Deleting one vertex reduces its arboricity down to two. The feedback vertex set of a K_5-graph contains three vertices. The solution for Arboricity-two-reduction of a K_5-graph contains one vertex.

further studied problem, vertex deletion to bounded degeneracy, is classified as W[P]-hard [15]. The relation between W[P] to FPT resembles to the relation between NP to P. The two parameters degeneracy and arboricity have the following relation:

$$\deg \leq \text{arb} \leq 2 \cdot \deg -1.$$

We observe that the arboricity of a graph differs from its degeneracy always at most by factor of two. There exist a large number of graph problems that are efficiently solvable for graphs with low degeneracy [6, 10, 18]. We can determine the degeneracy of a graph in linear time [13, 16]. But there is no known efficient way to check whether one can reduce the degeneracy of a given graph to a specific value by deleting at most k vertices [15]. We will show in our work that we can efficiently reduce the arboricity of a graph to two by deleting at most k vertices in FPT-time with parameter k. Thus reducing the arboricity of a graph in order to reduce its degeneracy in the process seems like a promising approach. Both of those graph parameters have a strong correlation to another, but there only exists an efficient way of reducing one of those parameters. Hence, to solve graph problems on an input graph, it is interesting to determine how far the input graph is away from a low degeneracy.

Algorithms already exist for Arboricity-zero-reduction, commonly known as Vertex Cover and Arboricity-one-reduction, also known as Feedback Vertex Set. Our algorithm delivers a foundation to extend the generalized problem in the next step by adding Arboricity-two-reduction to the list of explored problems in the problem class Arboricity-p-reduction, for any $p \in \mathbb{N}$. With this work we additionally aim to find patterns to eventually come up with an algorithm for the general case.

1.1 Related work

Yixin Cao [1] delivers an algorithm to solve the Feedback Vertex Set problem. The Feedback Vertex Set is a different term for arboricity-one-reduction, hence it is strongly connected to Arboricity-two-reduction. Cao's algorithm will serve as an inspiration for building an algorithm for Arboricity-two-reduction, in which we will use similar ideas and modify them accordingly to be applicable for Arboricity-two-reduction. Luo et al. [14] provide a similar algorithm to process *collapsing-k-cores*. The algorithm for this problem decides whether we can delete at most x vertices such that the largest k-core[1] of the remaining graph has size at most b. This problem is a generalized version of the problem class that Feedback Vertex Set falls under. Specifically, Feedback Vertex Set ist equivalent to *collapsing-k-cores* for $k = 2$ and $b = 0$. Mathieson and Luke [15] show the W[P]-hardness of vertex deletion to bounded degeneracy. As previously mentioned, this discovery motivates finding an efficient Arboricity-two-reduction algorithm because of the strong relation between arboricity and degeneracy.

1.2 Our contribution

In our work we analyse the computational complexity of Arboricity-two-reduction as fixed-parameter tractable for parameter k and deliver an FPT-algorithm to solve it. We also provide another algorithm, which is used as a subroutine for Arboricity-two-reduction, that solves Arboricity-two-reduction for maximum-degree-four graphs in polynomial time. This algorithm is not only relevant as a subroutine to build an FPT-algorithm for Arboricity-two-reduction but also is of independent interest. For an input graph with maximum-degree four, we do not have to call the entire Arboricity-two-reduction algorithm and solve the instance in FPT-time but can instead only call the subroutine and compute the solution in polynomial time. We also deliver groundwork to extend the problem to Arboricity-p-reduction to advance in eventually coming up with an algorithm for the general case for any $p \in \mathbb{N}$.

1.3 Organization of the work

We contine this work by introducing graph theory notation and parameterized complexity notation (see Chapter 2), which will be used throughout this work. In Chapter 3 we introduce Feedback Vertex Set and show its relation to Arboricity-two-reduction. We provide the reader with an algorithm (Algorithm 1) by Cao [1], that solves Feedback Vertex Set in FPT-time, which we briefly discuss. We also include an execution example to visualize how the algorithm finds its solution to clarify the greedy-branching method which we will adapt for our problem. In Chapter 4 we introduce Arboricity-two-reduction and explain how we can modify the existing

[1]The k-core is the largest subgraph with minimum degree k.

Feedback Vertex Set algorithm to be applicable to Arboricity-two-reduction. We then deliver an algorithm for Arboricity-two-reduction which presents those modifications. The algorithm (Algorithm 2) calls two subroutines, that mark the recursion base case, which will be discussed in Section 5.1 (Algorithm 3) and Section 5.2 (Algorithm 4). The first subroutine in Section 5.1 solves Arboricity-two-reduction for maximum-degree-four graphs in polynomial time. We explain how we came up with the algorithm and how it functions. In Section 5.2 we discuss the second part of the base case. In this algorithm we have to process vertices, that have not been properly considered in the previous step of the base case. We explain which vertices we have to consider to delete and which vertices we can ignore. Our algorithm for this step has to "guess" which of the considered vertices we have to delete by trying out all possible combinations. In the next chapter (see Section 6.1) we prove the correctness of the Arboricity-two-reduction algorithm and build a foundation to be able to classify the algorithm as fixed-parameter tractable. In Section 6.2 we conclude our results to analyse the running time, in which we prove that the Arboricity-two-reduction algorithm is indeed fixed-parameter tractable. We seperately show that our first subroutine Arboricity-two-reduction for maximum-degree-four graphs runs in polynomial time. Finally, we conclude our results in Chapter 7 and discuss interesting topics for future work (see Section 7.1).

2 Preliminaries

Graph theory. We use standard notation and terminology from graph theory [4]. Let $G = (V, E)$ denote an undirected graph, where V denotes the set of vertices and $E \subseteq \{\{v, w\} \mid v, w \in V, v \neq w\}$ denotes the set of edges. For a graph G, we also write $V(G)$ and $E(G)$ to denote the set of vertices and the set of edges of G, respectively. We define the cardinality of $V(G)$ as n_G and the cardinality of $E(G)$ as m_G. For a vertex set $T \subseteq V(G)$, we define $G[T]$ as the induced subgraph of G containing only the vertices of T. For a vertex v we define $G - \{v\}$ as the induced subgraph $G[V(G) \setminus \{v\}]$. For a set V' we define $G - V'$ as the induced subgraph $G[V(G) \setminus V']$. Correspondingly for a subgraph $G[V_1]$ and a vertex $v \in V(G)$ we define $G[V_1] + \{v\}$ as the induced subgraph $G[V_1 \cup \{v\}]$. For a graph G with two subgraphs $G[V_1]$ and $G[V_2]$ we define $G[V_1] - G[V_2]$ as the induced subgraph $G[V_1 \setminus V_2]$. Similarly we define $G[V_1] + G[V_2]$ as the induced subgraph $G[V_1 \cup V_2]$. The neighbors of a vertex v (also called neighborhood) are the set of vertices that v is connected to by an edge. The cardinality of the neighborhood for a vertex v is called degree of v or $\deg(v)$. A *d-regular* graph is a graph where every vertex has degree d.

The *arboricity* $\text{arb}(G)$ is defined as the minimum numbers of forests into which its edges can be partitioned. The formula for the arboricity of a graph G is $\text{arb}(G) = \max_{S \subseteq V(G)} \lceil \frac{m_{G[S]}}{n_{G[S]}-1} \rceil$. For a forest the amount of edges is at most $n - 1$. Hence, the arboricity of a graph is lower bounded by $\lceil \frac{m_G}{n_G-1} \rceil$. To get the exact value we have to look at every subgraph $G[S]$ and calculate the maximum $\lceil \frac{m_{G[S]}}{n_{G[S]}-1} \rceil$. Since this specific subgraph cannot be partitioned by less forests, the entire forest cannot be partioned by less forests.

A *d-degenerate* graph is an undirected graph in which every subgraph has a vertex of degree at most d. The *degeneracy* of a graph is the smallest value of d for which it is *d-degenerate*.

A *feedback vertex set* of a graph G is a vertex set X, for which holds that $G - X$ is a forest.

Parameterized complexity. We use basic notations from parameterized complexity and algorithmics [2, 5, 8, 17]. A problem is fixed-parameter tractable for parameter k, if it can be solved in $f(k) \cdot p(|I|)$ time, where I is the input instance, f is a computable function, and p is a polynomial. When a parameterized problem is W[P]-hard for a parameter k, it presumably does not admit an FPT-algorithm [2, 5, 8, 17].

3 Feedback Vertex Set

We now look at an algorithm that solves the Feedback Vertex Set problem, which will function as an inspiration for an algorithm for the problem we will propose in this thesis. The feedback vertex set of a graph G is the smallest vertex set, such that after removing those vertices the resulting graph is a forest. Note that a graph has arboricity one if and only if it is a forest. Hence Feedback Vertex Set is a different name for vertex deletion to arboricity one. Our goal in this thesis is to modify this algorithm to be compatible for vertex deletion to arboricity two instead of vertex deletion to arboricity one. We will modify the problem definition of Feedback Vertex Set to be easily changeable to our desired problem.

Feedback Vertex Set
Input: An undirected graph G and an integer $k \in \mathbb{N}$.
Question: Can the graph be reduced to arboricity one by deleting at most k vertices?
Output: A feedback vertex set $X \subseteq V(G)$ of size at most k or "NO".

We present an algorithm for the Feedback Vertex Set due to Cao [1] (for Pseudocode see Algorithm 1) as an inspiration for an algorithm for Arboricity-two-reduction. We will use following notation.

Algorithm-specific notation. The parameter k denotes the size of the desired solution, more precisely, whether we can reduce to arboricity two by removing at most k vertices. It basically transforms the algorithm from an optimization algorithm with the objective to minimize k to a decision algorithm that checks whether there exists a solution of size at most k. We define $X \subseteq V(G)$ as the vertex deletion set, which stores the vertices of the graph that are deleted from G to reduce its arboricity. Further we define an undeletable vertex set $Q \subseteq V(G)$ out of which our solution cannot contain any vertices.

The algorithm of Cao [1] (for Pseudocode see Algorithm 1) for the Feedback Vertex Set uses a greedy-branching approach, where the maximum-degree vertex at selection time is recursively partitioned into either the vertex deletion set X or the undeletable vertex set Q. The algorithm returns the vertex deletion set X with $|X| \leq k$ or "NO" if no such solution exist. It is obvious that each vertex can only be stored in at most one of the two sets.

Algorithm 1 Feedback Vertex Set Algorithm

Input: undirected graph G, parameter $k \in \mathbb{N}$, empty undeletable vertex set Q.
Output: either solution set X of size at most k, or "NO".

```
1:  function FVS(G, k, Q)
2:      if k < 0 or |Q| > 3k then return "NO";                ▷ Return conditions
3:      if V(G) = ∅ then return ∅
4:      for each v ∈ V(G) with deg(v) < 2 do                  ▷ Data reduction rules
5:          return FVS(G − {v}, k, Q \ {v})
6:      for each v ∈ V(G) \ Q with two neighbors in the same component of G[Q] do
7:          X ← FVS(G − {v}, k − 1, Q)
8:          return X ∪ {v}
9:      Pick vertex v ∈ V(G) \ Q with highest degree
10:     if deg(v) = 2 then
11:         X ← ∅
12:         while there is a cycle C in G do
13:             take any vertex x in C \ Q
14:             add x to X and delete x from G                ▷ Cycle elimination
15:         if |X| ≤ k then return X
16:         else return "NO"
17:     X ← FVS(G − {v}, k − 1, Q)                            ▷ Recursive call: v ∈ X
18:     if X is not "NO" then return X ∪ {v}
19:     return FVS(G, k, Q ∪ {v})                             ▷ Recursive call: v ∈ Q
```

We now present Cao's algorithm to solve Feedback Vertex Set (see Algorithm 1). Later in this section we will show an execution example to visualize how the algorithm works.

Algorithm 1 is initialized with three inputs, a graph G, an integer k, and an empty undeletable vertex set Q. It outputs either a solution set X or "NO" if no such solution with $|X| \leq k$ exists. The algorithm works as follows. The recursion ends if one of the IF-cases in lines 2-3 is satisfied. We will discuss the return conditions in detail in Chapter 4 for our specific problem. In lines 4-5 the algorithm removes every vertex with degree less than two. These vertices can never be part of a cycle and thus can be avoided by any solution for a feedback vertex set. Then in lines 6-8 the algorithm checks for a cycle between two vertices in the undeletable vertex set Q and a vertex outside of Q. Since we cannot delete vertices from the undeletable vertex set by definition of Q, the algorithm removes the vertex outside of Q to break the cycle. Next, in lines 9-16 the algorithm checks whether the highest degree in the graph is two. If this is the case, then every vertex will have degree two since every degree zero and degree one vertex is removed by the data reduction rules in lines 4-5. This means that the remaining graph consists of only cycles. The algorithm then removes an arbitrary vertex from every cycle and checks in line 15 if the amount of deleted vertices X is at most k. If it is, the

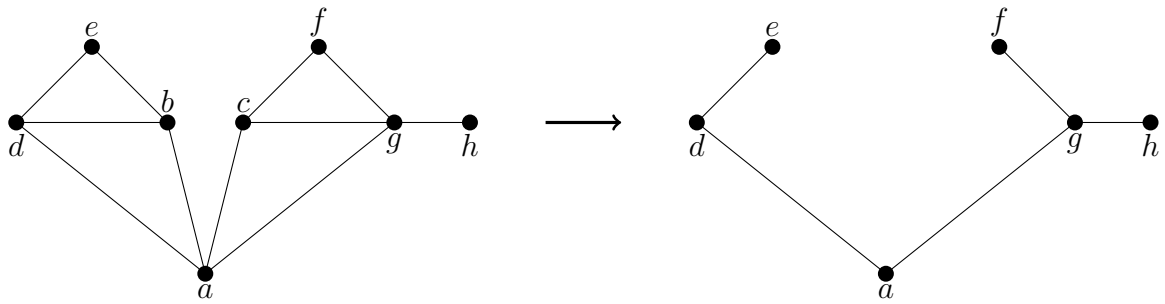

Figure 3.1: Example graph G. After removing one arbitrary vertex from both cycles the remaining graph is a forest.

algorithm returns X, otherwise it returns "NO". It is important that the algorithm can solve the Feedback Vertex Set of a maximum-degree-two graph in polynomial time to classify it as fixed-parameter tractable. We will show an FPT-time algorithm for the recursion base case of Arboricity-two-reduction in Chapter 5. In lines 17-19 the algorithm branches into two cases:

1. Either put v to the feedback vertex set, or
2. make v undeletable if no solution with v contained in the feedback vertex set exists.

In both cases the algorithm recurses. It is easy to observe that Q cannot contain a cycle, because any vertex that would lead to creating a cycle in Q is removed in lines 6-8. Hence, Q is a forest at all times. The algorithm then searches for a solution considering vertices in $G[V(G) \setminus Q]$. We repeat this process recursively until either a solution has been found or the algorithm terminates returning "NO".

We remark that termination and correctness of the algorithm are proven by Cao [1].

Execution Example. We now present an execution example of Algorithm 1 to display how the algorithm functions (see Figure 3.1). In the given graph the smallest feedback vertex set has size two. One can easily observe, that it has to contain one vertex from $\{b, d\}$ and one vertex from $\{c, g\}$ to delete all cycles in the graph. The remaining graph is a forest and thus does not contain any cycles.

In order to find a feedback vertex set of size two, we initialize the Algorithm 1 with the given input graph G (see Figure 3.1), $k = 2$, and an empty deletion set Q. The first step of the algorithm applies the data reduction rules. Hence, h is removed from G, because it has degree one. In the first iteration the algorithm does not enter the loop in lines 6-8 because Q is initially empty. Then the algorithm chooses the vertex with the highest degree, which is a. The algorithm now tries finding a solution with a in the solution set and thus deleted from the graph and $k = k - 1$, or $k = 1$. The algorithm now is in the second iteration. We are left with the following remaining graph (see Figure 3.2) after choosing to delete a from G. Since the highest degree is now two, the algorith jumps into the IF-condition in line 10. In lines 12-14 the algorithm deletes an arbitrary

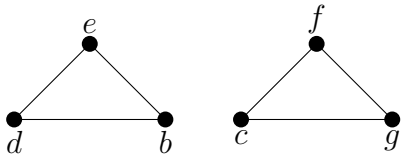

Figure 3.2: Remaining graph after applying the data reduction rules and removing a.

vertex from each cycle of the remaining graph. One can easily observe that there is no possible solution to eliminate both cycles by removing at most one vertex. Hence, after removing a vertex of both cycles and storing them in X, the size of X is larger than k. The algorithm returns "NO" in line 16. Since the algorithm returned "NO" for the second iteration, the algorithm now jumps into line 19 for the first iteration.

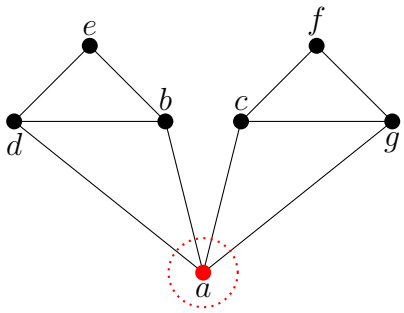

Figure 3.3: G after marking a as undeletable.

The vertex a is now marked as undeletable (see Figure 3.3) and the algorithm tries finding a solution with $(G, k = 2, Q = \{a\})$ in the second iteration. Again, the algorithm takes the vertex with the highest degree. Let this vertex be $d \in \{b, c, d, g\}$. The algorithm then jumps into the third iteration with $(G = G - \{d\}, k = 1, Q = \{a\})$. After applying the data reduction rules again we are left with the following remaining graph (see Figure 3.4).

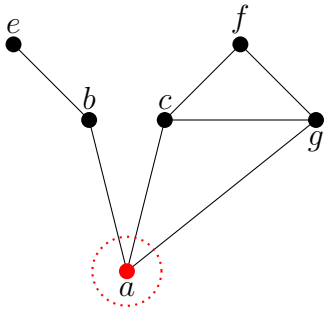

Figure 3.4: G after deleting d.

The algorithm once again applies the data reduction rules for the vertices e and b and then chooses a vertex with the highest degree. Let the vertex be $c \in \{c, g\}$. Removing c

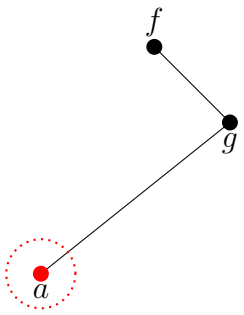

Figure 3.5: G after applying the data reduction rules and deleting c.

from the graph (see Figure 3.5), the algorithm jumps into the fourth iteration with the inputs $(G = G - \{c\}, k = 0, Q = \{a\})$. In the fourth iteration every remaining vertex will be removed by the reduction rules. Thus the algorithm jumps in line 3 and returns an empty solution set, implying it found a solution. Now the algorithm jumps in line 18 for each iteration and adds the deleted vertex to the solution set. In our example the algorithm returns $\{d, c\}$ and the algorithm is complete.

Note that if we would have chosen to find a solution for $k = 3$, the algorithm would have succeeded with taking a into our solution set. This solution would not have been minimal, but valid, since our algorithm is a decision algorithm and not an optimization algorithm.

4 Arboricity-Two-Reduction

As already discussed in Chapter 3, a greedy-branching algorithm works reliably for reducing the arboricity of a graph down to one. For our work we try to use a similar branching approach to reduce the arboricity of a graph to two by deleting vertices. Formally, we consider the following problem.

ARBORICITY-TWO-REDUCTION
Input: An undirected graph G and an integer $k \in \mathbb{N}$.
Question: Can G be reduced to arboricity two by deleting at most k vertices?
Output: A vertex deletion set $X \subseteq V(G)$ of size at most k or "NO"

In order for our algorithm (see pseudocode Algorithm 2) to work, we have to modify a few steps of the FEEDBACK VERTEX SET algorithm (Algorithm 1 [1]). As mentioned previously in Chapter 3, the termination of the Algorithm 1 in FPT-time is based on the condition that we can solve FEEDBACK VERTEX SET for a graph with maximum degree two in polynomial time. It is trivial that Algorithm 1 finds a solution in exponential time if it just tries out every possible solution. By bounding our return conditions by the parameter k, the algorithm finds a solution in FPT-time. Analogously to the FEEDBACK VERTEX SET, for the ARBORICITY-TWO-REDUCTION to reliably work in FPT-time we have to be able to stop the recursion and jump into a base case (see Chapter 5), where we solve ARBORICITY-TWO-REDUCTION FOR MAXIMUM-DEGREE-FOUR GRAPHS. We will show the reasoning for *maximum-degree-four* in the proof of the correctness of our algorithm (see Section 6.1).

4.1 Algorithm

We now propose our algorithm, a modification of Algorithm 1, to reduce the arboricity of a given graph G down to two by deleting at most k vertices. For the initial input we are given the graph G, a parameter k, and an empty set Q. Recall that Q contains the vertices that the algorithm marks as undeletable and thus cannot be included in our solution set X. This implies that $G[Q]$ must have arboricity at most two at all times. We will show this in Lemma 4.3. The algorithm outputs either a solution set X if it finds a solution with $|X| \leq k$, or "NO" otherwise. To simplify our pseudocode, the algorithm may return either a string ("NO") or a set X.

Algorithm 2 Arboricity-two-reduction Algorithm

Input: undirected graph G, parameter $k \in \mathbb{N}$, empty undeletable vertex set Q.
Output: either solution set X of size at most k, or "NO".

```
1:  function a2r(G, k, Q)
2:      if k < 0 or |Q| > 5k then return "NO"      ▷ Return conditions, see Section 6.1
3:      if V(G) = ∅ then return ∅
4:      for each v ∈ V(G) do
5:          if deg(v) ≤ 2 then                      ▷ Data reduction rules, see Section 4.2
6:              return a2r(G − {v}, k, Q)
7:      pick v ∈ V(G) \ Q with max degree           ▷ see observation Section 4.2
8:      if d(v) ≤ 4 then
9:          X ← 34a2r(G − G[Q], k)                  ▷ see Algorithm 3
10:         return guess(G, k, Q, X)                ▷ see Algorithm 4
11:     X ← a2r(G − {v}, k − 1, Q)                  ▷ Recursive call: v ∈ X
12:     if X is not "NO" then return X ∪ {v}
13:     if arb(G[Q ∪ {v}]) ≤ 2 then
14:         return a2r(G, k, Q ∪ {v})               ▷ Recursive call: v ∈ Q
15:     else
16:         return "NO"                             ▷ v cannot be added to Q, see Lemma 4.3
```

4.2 Description of Algorithm 2

Before we explain the algorithm itself, we will first point out the two major differences to Feedback Vertex Set. We have different return conditions, where we return "NO" if Q surpasses size $5k$ and we have a different base case, which will be specifically discussed in Section 5.1 and Section 5.2.

Algorithm 2 is initialized with three inputs, a graph G, an integer k, and an empty undeletable vertex set Q. It outputs either a solution set X or "NO" if no such solution with $|X| \leq k$ exists. The algorithm consists of five parts. We order the steps by the time they are first called instead of ordering them strictly by their lines.

1. The first part checks the return conditions (lines 2-3).

2. The second part deletes vertices from the graph which do not have to be considered for a solution (lines 4-6).

3. The third part checks recursively if there exists a solution containing the highest degree vertex (lines 11-16).

4. The fourth part marks the base case of the recursion. It is split up into two subroutines (lines 9-10).

4.1. The first part of the base case processes the graph in polynomial time after it has been reduced to a maximum degree of four (line 9).

4.2. The second part of the base case processes the remaining vertices after the branching and first part of the base case has been completed (line 10).

Step 1: Return conditions. The return conditions for the recursive branching steps (see lines 2-3 of Algorithm 2) are almost the same as the ones for Feedback Vertex Set. The difference is the "NO"-case when our undeletable vertex set Q becomes too large. We now return "NO", when the size of the undeletable vertex set surpasses $5k$ instead of $3k$. This value is not intuitive and will be explained later (see Section 6.1).

Step 2: Data reduction rules. The next difference to Algorithm 1 are the data reduction rules in lines 4-6. Like Algorithm 1, Algorithm 2 removes all vertices with degree zero and one from the graph.

Lemma 4.1. *Let (G, k) be an instance of* Arboricity-two-reduction *and $v \in V(G)$ a vertex with degree zero or one. Then (G, k) is a YES-instance if and only if $(G - \{v\}, k)$ is a YES-instance.*

Proof. Degree-zero and -one vertices can never form a cycle and thus can be avoided in any solution. □

But in contrast to Algorithm 1, Algorithm 2 also removes all vertices with degree two.

Lemma 4.2. *Let (G, k) be an instance of* Arboricity-two-reduction *and $v \in V(G)$ a vertex with degree two. Then (G, k) is a YES-instance if and only if $(G - \{v\}, k)$ is a YES-instance.*

Proof. Since we accept graphs with arboricity two, we know by definition of arboricity that there exists an edge partitioning such that the induced subgraph over the edges represents two forests F_1 and F_2, covering all vertices of G. If a vertex has degree two, we can partition both of its edges into a different forest. Thereby we essentially created two degree one vertices for either forest and the vertex can simply be removed (see Lemma 4.1). □

Observation for Step 3. Recall the definition of arboricity: $\text{arb}(G) = \max_{S \subseteq V(G)} \lceil \frac{m_{G[S]}}{n_{G[S]}-1} \rceil$. We observe that there is a strong connection between a high edge count and a high arboricity and vice versa. Thus it makes sense to consider putting our maximum degree vertex into our solution first, since it removes the most edges in the process.

Step 3: Branching Following our observation, Algorithm 2 picks the vertex v with the highest degree at selection time in line 7. If the degree is greater than four, then we recurse over two possible solution options. We check recursively if we find a solution with at most k vertices by taking v into our solution set in line 11. In the following

lemma we show that calling Algorithm 2 can never increase the arboricity of Q to three or higher. It is obvious to see that Q should never have arboricity greater than two, since we cannot delete vertices from Q. This lemma shows that adding a vertex to Q in line 14 does not violate this requirement.

Lemma 4.3. *$G[Q]$ has arboricity at most two at all times.*

Proof. We can check the arboricity of a graph G in $O(m_G^2)$ time [9]. Since we initialize Algorithm 2 with an empty deletion set, the lemma holds initially. We now have to show that after calling Algorithm 2 the arboricity of Q is never increased to more than two. Since we check in line 13 if the arboricity of $G[Q \cup \{v\}]$ is at most two, we only add a vertex to Q if it does not increase the arboricity of $G[Q]$ to more than two in the process. □

We now look at the two recursion steps that Algorithm 2 calls in lines 11 and 12.

- If we *do* find a solution with $v \in X$, then the algorithm will backtrack in line 12 and add every vertex chosen in the recursion to our solution.
- If we *do not* find a solution with $v \in X$, then there are two possible case distinctions. If $\text{arb}(G[Q \cup \{v\}])$ is at most two, then we put v into Q and call Algorithm 2 with the arguments $(G, k, Q \cup \{v\})$. Otherwise, we return "NO". From Lemma 4.3 we know that adding v to Q does not increase the arboricity of Q to three.

If we do find a solution, then the correctness and termination of this greedy-branching approach is trivial (assuming our return coniditions, upper bounded by k, are correct, proof in Section 6.1). By recursively trying out each possible solution (combined with the base case) and backtracking when we jump into the return conditions, we guarantee terminating eventually either with a solution or "NO" if no solution of size k exists. Remember that our algorithm does not solve an optimization problem but a decision problem. Thus we can correctly terminate the algorithm after finding the first solution with size $\leq k$. Consequently, if there exists a solution with size at most k, then Algorithm 2 will find it. We will show later in the proof, that we cannot find a solution if our set becomes too large, meaning it is not necessary to branch through the entire tree (see Section 6.1).

The algorithm jumps in step 4, when every remaining vertex in the graph has degree less than four (see line 8).

Step 4: Base Case. We now look at the two subroutines that are called in the recursion base case. At this point the data reduction rules have been performed, thus the remaining graph has vertices of degree three and four. We introduce a new name for those specific graphs, as they will be used throughout this work.

Definition 4.4. A graph where every vertex has degree either three or four is called *3-4-graph*.

The reasoning behind stopping the recursion early and jumping in a base case will be shown in Section 6.1.

Step 4.1: Arboricity-two-reduction for maximum-degree four In Section 5.1 we discuss how to solve Arboricity-two-reduction on 3-4-graphs in polynomial time. The core idea of this algorithm is that we partition the subgraphs into categories and process only the subgraphs for which $\lceil \frac{m}{n-1} \rceil = 3$. This will be further discussed in detail in Section 5.1.

Step 4.2: Processing vertices connected to $G[Q]$. The second subroutine (see Section 6.1) is called by in line 10, after the first step of the base case has been processed. This algorithm processes the vertices that connect from the remaining graph to $G[Q]$. The first part of the base case ignores the edges from a vertex in the remaining graph to a vertex in $G[Q]$ and thus cannot check if they increase the arboricity. The Feedback Vertex Set (see Algorithm 1) solves this problem implicitely in lines 6-8. For Arboricity-two-reduction this step is not as trivial, since we cannot easily tell whether we have to delete a vertex. Hence, we have to try out all combinations of deleting a vertex.

In the next chapter (see Chapter 5) we discuss the base case of Algorithm 2. We propose an algorithm for both of the subroutines and discuss them. We will also show how we built the algorithm and why specifically the first subroutine, where we solve Arboricity-two-reduction for maximum-degree-four graphs, is of independent interest.

5 Recursion Base Case

This chapter will dicuss the two subroutines that Algorithm 2 calls in lines 9-10. We present an algorithm for both problems, for which we will proof their correctness. In the following section we discuss how we built an algorithm to process 3-4-graphs. In the remark at the end of this section we explain how we can generalize the algorithm to apply for all graphs with maximum-degree four.

5.1 Arboricity-Two-Reduction for Maximum Degree Four

In this section we present an algorithm running in polynomial time which solves Arboricity-two-reduction for 3-4-graphs. We assume that we have preprocessed every vertex with degree at most two in lines 4-6 of Algorithm 2 and all vertices with degree higher than four in lines 11-16, before the following algorithm (see Algorithm 3) is called in line 9 of Algorithm 2.

Arboricity-two-reduction for 3-4-graphs
Input: An undirected 3-4-graph G and an integer k.
Question: Can G be reduced to arboricity two by deleting at most k vertices?
Output: A vertex deletion set $X \subseteq V(G)$ of size at most k or "NO".

The main result of this chapter is to show, that we do not need to recursively try out solutions when the maximum degree in the graph is at most four. We will present an algorithm that directly computes a solution in polynomial time, if a solution exists. Otherwise it will return "NO".

Theorem 5.1. Arboricity-two-reduction *for 3-4-graphs can be solved in polynomial time.*

The main idea of our algorithm is to partition the input graph into several subgraphs and process them individually. We will first investigate in which pairwise relation two subgraphs can be, so we can then dissolve subgraphs with common vertices into a number of individual subgraphs with no common vertices. We then classify our subgraphs to decide for each one individually whether we have to delete a vertex of it or not. We will show that after processing every subgraph the whole graph will have arboricity two.

Definition 5.2. The *set density* $\mathrm{sd}(G)$ of a graph G is $\lceil \frac{m_G}{n_G - 1} \rceil$.

Recall that the definition of the arboricity of a graph G is $\text{arb}(G) = \max_{S \subseteq V(G)} \text{sd}(G[S])$. The formula returns the highest set density of every subgraph of G. So in order to reduce the arboricity of a graph down to two, we have to reduce the set density of every single subgraph of the graph to two or less. It is obvious that the arboricity of a graph G is lower-bounded by the set density of the entire graph G since the arboricity is the maximum set density of all subgraphs of G, including G itself.

We observe that a 3-4-graph has at most arboricity three. Hence we consider only the subgraphs with set density three. Those subgraphs are the ones that have to be modified, since we have to achieve a set density of at most two for every subgraph. We show in the following lemma that every 3-4-graph with set density at least three contains at least one degree four vertex.

Lemma 5.3. *A 3-4-graph with set density three contains at least one degree four vertex.*

Proof. Assume for contradiction that we have a 3-4-graph S with set density three and only degree-three vertices. Hence the graph is 3-regular and thus contains at least four vertices. This means that the amount of edges m_S is $\frac{3}{2}|n_S|$. We then have

$$\text{sd}(S) = \lceil \frac{m_S}{n_S - 1} \rceil = \lceil \frac{3n_S}{2n_S - 1} \rceil < 3, \text{ for } n_S \geq 4.$$

This contradicts the assumption that S has set density three, hence the proof is complete. □

This allows us to always remove a degree four vertex from a 3-4-graph with set density three. We will show in the following lemma that removing such degree four vertex will decrease the set density from three to two.

Lemma 5.4. *Removing a single degree four vertex of a 3-4-graph with set density three will reduce its set density to two.*

Proof. We observe that the set density gets smaller by removing edges. Hence a graph with the maximum number of edges, a 4-regular graph, will be our worst case, which we aim to modify down to set density two. We observe that a 4-regular graph has set density three. Note that a 4-regular graph has at least five vertices.

$$\text{sd}(S) = \lceil \frac{m_S}{n_S - 1} \rceil = \lceil \frac{2n_S}{n_S - 1} \rceil = 3, n_S \geq 5.$$

Now we remove a degree four vertex: In that process four edges get removed as well. For the reduced graph S' we obtain a new set density of

$$\text{sd}(S') = \lceil \frac{2n_S - 4}{n_S - 2} \rceil = 2.$$

Since every 3-4-graph has at most $2|V(G)|$ edges, removing a single degree-four vertex from any graph with set density three will always reduce its set density down to two. □

We now classify our graphs in different categories to decide whether a given graph or subgraph needs to have vertices deleted. Recall that every subgraph has to have a set density less than three for the whole graph to have arboricity two.

Definition 5.5. A graph S is *minimal* when $\forall S' \subset S : \mathrm{sd}(S') < \mathrm{sd}(S)$.

This means that if a graph S is minimal, then the set density of every subgraph of S is smaller than the set density of S itself.

Definition 5.6. Graphs with set density three are called *problematic graphs*.

Problematic graphs are the subgraphs that have to be modified for the whole graph to obtain arboricity two.

Definition 5.7. Graphs with set density two are called *potentially problematic graphs*.

It is important to note that a set density of two does not imply an arboricity of two. A graph T might have a set density of two, but a subgraph T' of T might have a set density of three, resulting in arboricity three for the graph T. Hence, potentially problematic graphs have to be further observed and modified if they are not minimal.

Definition 5.8. Graphs with arboricity two and set density two are called *unproblematic graphs*.

Unproblematic graphs will not be modified in the algorithm since they cannot cause the graph to exceed arboricity two.

According to Lemma 5.4 we have to remove a vertex from all problematic subgraphs for the whole graph to have arboricity two. The following lemma makes a small observation that allows us to determine how to process the problematic subgraphs.

Lemma 5.9. *If a solution exists, then there always exists a solution for that every vertex out of a solution is contained in a minimal subgraph.*

Proof. We first show a naïve approach how we can solve all problematic subgraphs by splitting them up into only minimal subgraphs. This lemma will be used to prove that we never have to process components which are not minimal. Problematic subgraphs can appear in three different relations. Consider the following pairwise relations of two problematic subgraphs S_1 and S_2:

1. $S_1 \subset S_2$

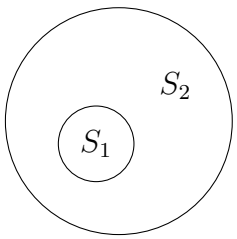

Figure 5.1: All vertices of S_1 are vertices of S_2, but not vice versa. S_2 is not minimal.

2. $S_1 \cap S_2 = S_3 \neq \emptyset, S_1 \neq S_3 \neq S_2$

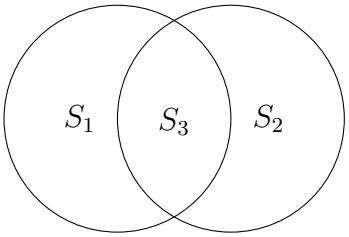

Figure 5.2: S_1 and S_2 have common vertices, inducing the graph S_3. S_1 and S_2 are not minimal.

3. $S_1 \cap S_2 = \emptyset$

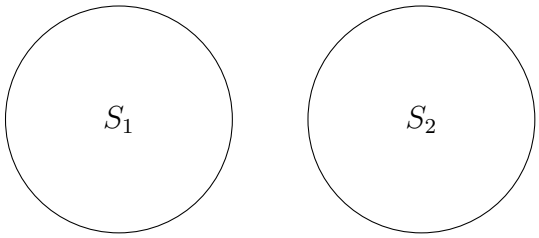

Figure 5.3: S_1 and S_2 have no common vertices. Both S_1 and S_2 are minimal.

Now we show how to process the problematic subgraphs for each of the three cases. We will split up every case into an instance containing only minimal subgraphs, which we can then solve individually.

Case 1 (see Figure 5.1). We assume that Cases 2 and 3 do not apply. For every Case-1-instance we split up S_1 and S_2 into S_1 and $S_2' = S_2 - S_1$.

Case 2 (see Figure 5.2). We assume that Case 1 does not apply. For Case 2 we repartition the subgraphs S_1 and S_2 into the following new subgraphs:

1. $S_1' = S_1 - S_3$
2. $S_2' = S_2 - S_3$

By definition we know that $S_3 \neq \emptyset$. Hence $S_1' \subset S_1$ and $S_2' \subset S_2$. We have effectively split up the Case-2-instance into an instance of Case 3 with three minimal subgraphs S_1', S_2', and S_3.

Case 3 (see Figure 5.3). If no Case-1 or Case-2-instances exist, we are left with only minimal subgraphs.

We now need to prove that our new solution, where every vertex is part of a minimal subgraph, is indeed correct. Assume we have a problematic and not minimal subgraph S. Now we prove that when we split up S into problematic minimal subgraphs and a restgraph and then *only* process the minimal subgraphs, then S will be unproblematic. As we have seen, we can always split up a problemtatic graph S in minimal problematic subgraphs $S' = \{S'_1, S'_2, \ldots, S'_\ell\}$ and an unproblematic remaining graph $S^\star$. We then applied Lemma 5.4 on every subgraph in S', hence every subgraph will be unproblematic after processing. The arboricity is the maximum set density of every subgraph. Since $S^\star$ is unproblematic it has arboricity two and set density two. After processing every subgraph of S has set density two and thus S has arboricity two. To conclude, we showed that we can solve an instance of every case by only modifying minimal subgraphs. The order of modifying minimal subgraphs does not matter, since modifications on a minimal subgraph do not influence any other minimal subgraph. Thus Lemma 5.9 is correct. □

We now define *outgoing* edges, to use them in the following lemma (see Lemma 5.11).

Definition 5.10. An edge that connects a vertex from a subgraph A to a vertex outside of A is called *outgoing edge* with respect to A.

Lemma 5.11. *A 3-4-graph with at least two outgoing edges is unproblematic.*

Proof. Recall the definition of unproblematic graphs (Definition 5.8). Consider a 4-regular graph S with at least five vertices. Since every vertex has four outgoing edges and each edge is counted twice, the amount of edges we have in 4-regular graphs is $2|V(S)|$. Thus we can transform our set density formula to the following: $\mathrm{sd}(S) = \lceil \frac{2n_S}{n_S-1} \rceil = 3$. What we can observe is that if S is missing two edges (equivalent to having two or more outgoing edges, since outgoing edges are not counter in m_S) we would yield $\lceil \frac{2n_S-2}{n_S-1} \rceil = 2$. □

Assume that we have a set U of all maximal 2-edge-connected components of G. To understand what maximal 2-edge-connected graphs are we introduce some vocabulary. We first explain what connectivity means and what components are. A graph is *connected* when there is a path between every pair of vertices. A graph is *disconnected* if it is not connected. A *component* is a connected subgraph. From this we can define what 2-edge-connected graphs and 2-edge-connected components are.

Definition 5.12. A graph or component is *2-edge-connected* if you have to remove at least two edges to disconnect it.

Since we specified that our set U consists of all *maximal* 2-edge-connected components, we have to define when a 2-edge-connected component is *maximal*.

Definition 5.13. A 2-edge-connected component C is *maximal* if there exists no other 2-edge-connected component of which C is a proper subgraph.

Recall that U stores all maximal 2-edge-connected components. In our following lemma we show that U includes all potentially problematic subgraphs.

Lemma 5.14. *Minimal problematic subgraphs are maximal 2-edge-connected components.*

Proof. For this proof we have to show the two following statements:

1. A component C is problematic and not 2-edge-connected. $\Rightarrow C$ is not minimal.
2. A 2-edge-connected component C is not maximal. $\Rightarrow C$ is not problematic (from Lemma 5.11).

First of all we show that a graph that is problematic and not a 2-edge-connected component is not minimal. Recall the definition of minimal (see Definition 5.5). Assume for contradiction that C is a minimal problematic subgraph and not 2-edge-connected. When a graph is not a 2-edge-connected component, it can be disconnected into two components C_1 and C_2 by removing at most edge. Since C is also minimal both C_1 and C_2 are not problematic. Adding a single edge between two unproblematic subgraphs cannot raise the arboricity of the entire graph C. Hence, C cannot be problematic which proves that a graph cannot be minimal, problematic and not 2-edge-connected at the same time. Now we show by contradiction that the second statement is true. Assume we have a problematic 2-edge-connected component C which is not maximal. Recall that every vertex of C has a maximum degree of four, since we consider C in the base case. In the following lemma we show that C is a 3-4-graph.

Claim 5.16. *A problematic graph with maximum degree four is a 3-4-graph.*
In the following we proof Claim 5.16. Assume for contradiction that C is a problematic graph with maximum degree four, but not a 3-4-graph. Hence, C must have at least one vertex with degree less than three. We know from Lemma 5.11 that if a graph has at least two edges less than a K_4 graph, then the graph is unproblematic. Since C has a vertex with degree less than three, C must be unproblematic. This contradicts the assumption that C is problematic and thus Claim 5.16 is correct.

When a 2-edged-component C is not maximal, then there exists another 2-edge-connected component C', with C being a subgraph of C'. Since C' is 2-edge-connected, there are at least two outgoing edges from C to C' (recall the definition of outgoing edges Definition 5.10). From Lemma 5.11 and Claim 5.16 it follows that C must be unproblematic, proving the contradiction. Combining both of these statements we show that Lemma 5.14 is correct. □

Now we have all necessary ingredients to show the correctness of Theorem 5.1.

Proof of Theorem 5.1. To recap, we can enumerate all maximal 2-edge-connected components [7]. We know by Lemma 5.14 that they contain all minimal problematic subgraphs. By Lemma 5.9 we know that we only need to modify those subgraphs. After

Algorithm 3 ARBORICITY-TWO-REDUCTION FOR 3-4-GRAPHS ALGORITHM

Input: undirected 3-4-graph G, parameter $k \in \mathbb{N}$.
Output: either solution set X of size at most k, or "NO".

```
1:  function 34a2r(G, k)
2:      find maximum 2-edge-connected components and store in set U
3:      while U not empty do
4:          take any subgraph P ∈ U
5:          if P is problematic then
6:              remove maximum degree vertex from P and put it into X
7:      if |X| ≤ k then
8:          return X
9:      else
10:         return "NO"
```

being left with only minimal graphs we can apply Lemma 5.4 to reduce the arboricity of a graph with maximum degree four in polynomial time. □

To conclude, we sum up the steps into an algorithm (see Algorithm 3). The algorithm is called in line 9 of Algorithm 2 with the arguments $(G-G[Q], k)$. We only consider $G-G[Q]$ since we aim to modify the remaining 3-4-graph and thus ignore the vertices of Q. This means that edges from a vertex in the remaining graph to $G[Q]$ are ignored by Algorithm 3. We will show in Section 5.2 how we seperately process the vertices in the remaining graph that connect to a vertex in $G[Q]$. The second input is k, which we need to decide whether Algorithm 3 returns the solution set or "NO". The algorithm works as follows. First we need to find all 2-edge-connected components and store them in a set U (line 2). We know from Lemma 5.9 that all other subgraphs are unproblematic and thus we do not delete a vertex out of them. For every problematic subgraph out of U (lines 4-5) we delete the maximum degree vertex (justified by Lemma 5.4) and add it to our solution set X (line 6). When all problematic subgraphs are processed, then our algorithm is finished and returns the set X of deleted vertices if $|X| \leq k$, or "NO" otherwise (lines 7-10).

Remark: Algorithm 3 solves ARBORICITY-TWO-REDUCTION for 3-4-graphs. For input instances with maximum degree four, we can extend the algorithm to reapply the data reduction rules beforehand (see Section 4.2). This allows us to independently call this algorithm to solve input graphs with maximum-degree four in polynomial time.

5.2 Processing vertices connected to $G[Q]$

In the first step of the base case we solved ARBORICITY-TWO-REDUCTION for the remaining 3-4-graph $G - G[Q]$. We only considered the vertices of the remaining graph, since vertices from Q do not necessarily have a maximum degree of four. Hence we need

another algorithm to process the vertices in the remaining graph that have an edge to a vertex from Q. In the following we will refer to those vertices as *connecting vertices*.

Definition 5.15. For two graphs we define a *connecting vertex* as a vertex from one set that has an edge to a vertex from the other set.

In this section we discuss how to process the connecting vertices from the remaining graph to the undeletable set Q. In the following, we will refer to the vertex set of the remaining graph $V(G) \setminus Q$ as F. We assume that if we get to this stage, we have performed the base case Algorithm 3, hence both $G[Q]$ and $G[F]$ have an arboricity of two. We know that F is a 3-4-graph, because we applied the reduction rules in Algorithm 2 beforehand and only consider graphs with maximum degree four for the base case (see line 8-10 of Algorithm 2). Hence, there only exists a small number of possible relations a vertex from $G[F]$ can have to $G[Q]$. First we take a look at degree-four vertices. Our goal is to find a solution for those cases and to cover the cases for degree-three vertices in the process.

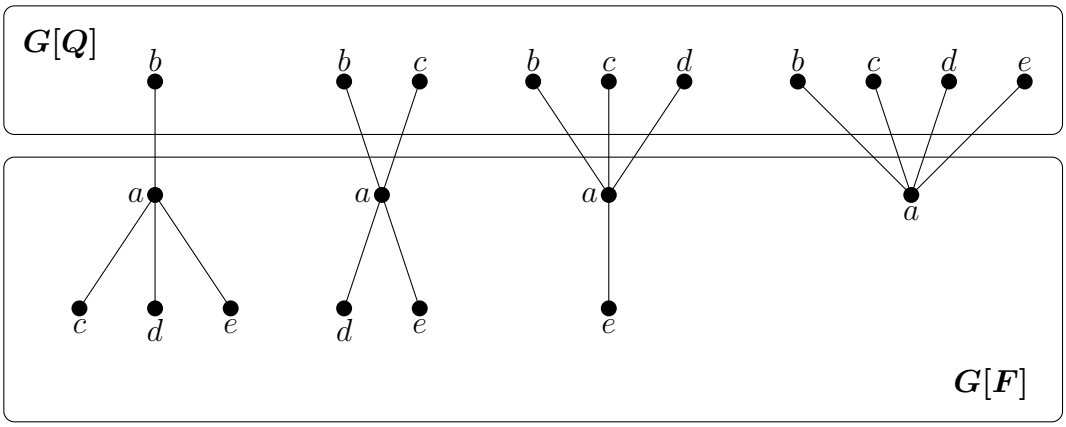

Figure 5.4: Relations that a degree-four vertex $a \in G[F]$ can have to $G[Q]$.

As we can see in Figure 5.4 there are four possible relations from a degree four vertex a to Q:

Case 1: a is connected to one vertex in Q and three vertices in F.

Case 2: a is connected to two vertices in Q and two vertices in F.

Case 3: a is connected to three vertices in Q and one vertex in F.

Case 4: a is connected to four vertices in Q and no vertex in F.

We first show how we process vertices that are classified in Case 3 or Case 4. We will show later in this section, that we do not have to modify a vertex at all if falls under the category of Case 1 or Case 2.

Algorithm 4 Processing vertices connected to $G[Q]$

Input: undirected 3-4-graph G, parameter $k \in \mathbb{N}$, undeletable set Q, solution set X.
Output: either solution set X of size at most k, or "NO".

```
1:  function guess(G, k, Q, X)
2:      if X == "NO" then
3:          return "NO"
4:      for v ∈ V(G) \ X do
5:          if (deg(v) == 3 or deg(v) == 4) and every neighbor of v in G[Q] then
6:              store v in Z
7:      partition Z into categories, such that one category corresponds to any
        combination of three vertices from G[Q] or four vertices from G[Q]
8:      for all combinations of deleting at most k−|X| vertices per category and deleting
        at most k − |X| vertices altogether do
9:          X ← X∪ deleted vertices
10:         if arb(G − G[X]) ≤ 2 then
11:             return X
12:     return "NO"
```

Case 4 is problematic and we cannot just ignore those vertices in our solution. We have to "guess" which vertices to delete. The same applies for vertices with degree three, that have no connection to a vertex from $G[F]$. The algorithm to process those vertices (for pseudocode see Algorithm 4) works as follows.

Algorithm for Case 4 We call Algorithm 4 in line 10 of Algorithm 2 with the arguments (G, k, Q, X). We know that the first step of the base case has been processed beforehand in line 9 of Algorithm 2 and the found solution set is stored in X. We call Algorithm 4 with X in the input, because it needs to be able to ignore vertices that Algorithm 3 has already marked as part of the solution. The algorithm outputs either the solution set X, or "NO" otherwise. The algorithm works as follows. If the algorithm is called with $X =$ "NO" then we know that Algorithm 3 has not found a solution. Hence, we can terminate early and return "NO" (lines 2-3). If the base case has indeed found a solution and thus Algorithm 4 is called with $X \neq$ "NO", then the algorithm iterates over all vertices from $G[F]$ (line 4). For every vertex with degree three or four it checks if all its neighbors are in $G[Q]$ (line 5). Every vertex that satisfies the IF-condition is stored in a vertex set Z (line 6). This set contains all vertices that are potentially part of the solution. This implies that every vertex for which the IF-condition is false does not have to be deleted. We show the correctness of this statement later in this section. In line 7 the algorithm partitions Z into categories. There is one category for each combination of three or four vertices in Q. We categorize Z because it does not matter *which* vertex we delete from a category, since deleting any vertex from a category modifies the graph in the same way. This allows us not having to test every possible permutation of which vertices we have to delete but rather just *how many* vertices we have to delete from a category. In line 8 the algorithm tries out all combinations of deleting 0 to $k - |X|$

vertices for each category, deleting at most $k - |X|$ vertices altogether. and checks the arboricity of the resulting graph for each combination. We can only delete a maximum number of $k - |X|$ vertices because Algorithm 3 has already put $|X|$ vertices into the solution set. Note that it does not matter *which* vertices are deleted out of a category, since all vertices out of a category have the same relation to Q. For every combination we add the set of deleted vertices to X and check if the graph has obtained arboricity two by deleting the vertices of X from G (lines 9-10). If Algorithm 4 finds a solution which satisfies the IF-condition, then it returns X (lines 10-11). Otherwise the algorithm returns "NO" (line 12). Note that we do not have to explicitely check if our solution has size at most k before returning it, since we only consider solutions in line 8 that have a total size of at most k. The termination of Algorithm 4 is trivial, since we try out every possibility.

Algorithm 4 obviously works for Case 4 instances, where the vertex has no connection to any other vertex in $G[F]$. In our following lemma we will prove that Algorithm 4 also solves the Case 3 instances (see Figure 5.4). We have to show that it does not matter to which vertex $e \in G[F]$ the considered vertex a is connected. This allows us to *only* partition our categories by the neighborhood of a in $G[Q]$. This is highly important for the running time of this algorithm, which we will discuss in Section 6.2. In Lemma 5.16 we show that adding a Case 3 vertex (Figure 5.4) does not increase the set density of either set. By proving Lemma 5.16 we show that we can categorize every Case 3 vertex by *only* their neighborhood in $G[Q]$.

We assume that Q has an arboricity of at most two (justified by Lemma 4.3) and $G[F]$ has an arboricity of at most two, because we have processed $G[F]$ in Algorithm 3.

Lemma 5.16. *For any two subgraphs $G[F']$ subgraph of $G[F]$ with $|F'| \geq 2$ and $G[Q']$ subgraph of $G[Q]$ with $|Q'| \geq 3$ with set density at most two for both subgraphs and a vertex $a \in F'$ with three edges to $G[Q']$ and one edge to $G[F']$, the set density of $G[F' \cup Q']$ can never be three (Case 3).*

Proof. Instead of viewing Q' and F' we split up the problem into Q' and $F' - \{a\}$ (see Figure 5.5). Since we know that F has arboricity two (see Section 5.1), then $F' - \{a\}$ has set density two. By removing a we remove one vertex and one edge from F'.

$$\lceil \frac{m_{F'-\{a\}}}{n_{F'-\{a\}} - 1} \rceil \leq 2 \quad \Rightarrow \quad \lceil \frac{m_{F'} - 1}{n_{F'} - 2} \rceil \leq 2 \tag{5.1}$$

Recall that we know for Q, that

$$\lceil \frac{m_{Q'}}{n_{Q'} - 1} \rceil \leq 2. \tag{5.2}$$

We now know that both Q' and F' have a set density of two without being connected to a. Now we need to show that if we add a back into our subgraphs, then $Q + F' + \{a\}$

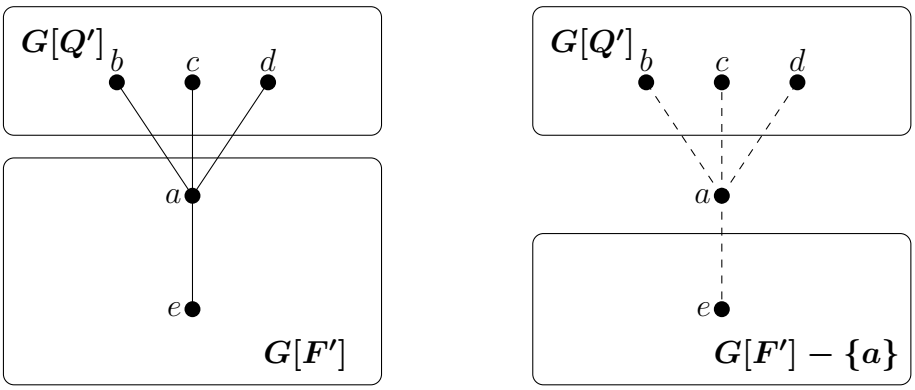

Figure 5.5: Reconstructing the problem to consider the relation between $G[Q']$ to $G[F'] - \{a\}$ instead of $G[Q]$ to $G[F']$.

will still have a set density of two. Hence by adding a to $Q + F'$ we add four edges and one vertex to the equation.

$$\lceil \frac{m_{F'-\{a\}} + m_{Q'} + 4}{n_{F'-\{a\}} + n_{Q'} - 1 + 1} \rceil \quad = \quad \lceil \frac{m_{F'-\{a\}} + m_{Q'} + 4}{n_{F'-\{a\}} + n_{Q'}} \rceil \quad \overset{!}{\leq} \quad 2. \tag{5.3}$$

We know from

$$\lceil \frac{m_{F'-\{a\}}}{n_{F'-\{a\}} - 1} \rceil \leq 2 \quad \text{and} \quad \lceil \frac{m_{Q'}}{n_{Q'} - 1} \rceil \leq 2, \quad \text{that} \quad \lceil \frac{m_{F'-\{a\}} + m_{Q'}}{n_{F'-\{a\}} + n_{Q'} - 2} \rceil \leq 2.^{1} \tag{5.4}$$

If we now expand our fraction by adding 4 to the numerator and 2 to the denominator, then we obtain

$$\lceil \frac{m_{F'-\{a\}} + m_{Q'} + 4}{n_{F'-\{a\}} + n_{Q'} - 2 + 2} \rceil \leq 2 \quad \Rightarrow \quad \lceil \frac{m_{F'-\{a\}} + m_{Q'} + 4}{n_{F'-\{a\}} + n_{Q'}} \rceil \leq 2. \tag{5.5}$$

Thus we have shown Equation (5.3) and the proof of Lemma 5.16 is complete. □

The last thing we have to show is that Case-1-instances and Case-2-instances are indeed unproblematic and we can ignore them in Algorithm 4. We only need to show that we can add the edges from $G[F]$ to $G[Q]$ (see Figure 5.4), since the edges within $G[F]$ have already been considered by Algorithm 3.

Lemma 5.17. *Adding at most two edges between any $G[F']$ subgraph of $G[F]$ with $|F'| \geq 1$ and any $G[Q']$ subgraph of $G[Q]$ with $|Q'| \geq 2$ does not increase the set density of $G[F' \cup Q']$ to three (Cases 1 and 2).*

[1] This follows from the fact that

$$\frac{a}{b} \leq 2 \text{ and } \frac{c}{d} \leq 2 \rightarrow \frac{a+c}{b+d} \leq \frac{2b+2d}{b+d} = 2.$$

Proof. We know from Lemma 4.3 that Q has an arboricity of at most two at all times. Hence, Q also has a set density of at most two. Let $F' \subseteq F$ be an arbitrary subgraph of F. We know that F' has a set density of two, since we view this situation after processing the first part of the base case. We have

$$\lceil \frac{m_{F'}}{n_{F'} - 1} \rceil \leq 2 \qquad \text{and} \qquad \lceil \frac{m_Q}{n_Q - 1} \rceil \leq 2. \tag{5.6}$$

From this it follows that

$$\lceil \frac{m_{F'} + m_Q}{n_{F'} + n_Q - 2} \rceil \leq 2. \tag{5.7}$$

The set density of $F' + Q$ is

$$\lceil \frac{m_{F'} + m_Q + 2}{n_{F'} + n_Q - 1} \rceil \qquad = \qquad \lceil \frac{(m_{F'} + m_Q) + 2}{(n_{F'} + n_Q - 2) + 1} \rceil \overset{(5.2)}{\leq} 2. \tag{5.8}$$

□

Degree-three vertices We mentioned earlier that we only explicitely look at degree-four vertices, since degree-three vertices with edges to both $G[F]$ and $G[Q]$ get implicitely solved in the process. We already showed how to process degree-three vertices with only edges to $G[Q]$ (see Algorithm 4). It is obvious that degree-three vertices with edges to both subgraphs only have one edge less in $G[F]$ and thus can be ignored by the same argument.

6 Analysis of Algorithm 2

In this chapter we will analyse the correctness of Algorithm 2 (see Section 6.1) and its total running time (see Section 6.2).

6.1 Correctness

We have already proven the correctness of the base case of Algorithm 2 in Chapter 5. What is left to do is to prove that the return conditions are indeed correct. We mentioned in Section 4.2 that our recursive call of Algorithm 2 returns "NO" if $|Q| > 5k$, which implies that there exists no solution when more than $5k$ vertices are stored in Q. In this section we discuss why the algorithm cannot find a valid solution if this set becomes too large and does not need to continue branching. To recap, the other return-"NO"-condition was if the size of the solution set X is bigger than k. This is trivial. We now formulate the Theorem that we aim to prove in this section.

Theorem 6.1. *If Q contains more than $5k$ vertices, then there is no solution containing all vertices from X and no vertex from Q.*

Proof. Assume for contradiction that there is a solution $X \subseteq X^\star \subseteq V$ with $|X^\star| \leq k$ and $X^\star \cap Q = \emptyset$, that is, $X^\star$ contains all vertices of X and no vertex from Q. We are going to produce a contradiction by a counting argument over the degrees of vertices in S.

Let B be $X \cup Q$, including all vertices in both sets. Let $>_{\text{select}}$ be the order in which Algorithm 2 selected the vertices in B and let $\deg^*(v)$ be the degree of $v \in B$ at the time of selection for the recursive calls of Algorithm 2, that is, $\deg^*(v) = \deg_{G-X_>(v)}(v)$, where $X_>(v) := \{v' \in X \mid v' >_{\text{select}} v\}$. It is easy to observe that $v >_{\text{select}} v'$ implies $\deg^*(v) \geq \deg^*(v')$. Let $N^*(v)$ denote the neighborhood of v at selection time. Now we take a closer look at the degree at selection time of a vertex in $v \in X$ (see line 11 of Algorithm 2). We define the following:

- $\deg^*_{X^\star}(v) := |N^*(v) \cap X^\star|$

  (Number of neighbors of v at selection time that are contained in the solution $X^\star$.)

- $\deg^*_{Q}(v) := |N^*(v) \cap Q|$

  (Number of neighbors of v at selection time that are contained in Q.)

- $\deg^*_{\text{remaining}}(v) := |N^*(v) \setminus (X^\star \cup Q)|$
  (Number of neighbors of v at selection time that are neither contained in the solution $X^\star$ nor in Q.)

It is obvious that $\deg^*(v) = \deg^*_{X^\star}(v) + \deg^*_Q(v) + \deg^*_{\text{remaining}}(v)$ which obviously implies that $\deg^*(v) \geq \deg^*_Q(v)$. To take a closer look at $\deg^*_Q(v)$ and distinguish between neighbors of v at selection time in Q (see line 14 of Algorithm 2) we need to introduce the function *chainend*.

Remark: Since we accept graphs with arboricity two, we know by definition of arboricity that there exists an edge partitioning, such that the induced subgraph over the edges represent two forests F_1 and F_2, covering all edges of G. We define $\deg_{F_1}(v)$ as the amount of neighbors of v in forest one. Analougously $\deg_{F_2}(v)$ denotes the amount of neighbors of v in forest two. We observe that $\deg(v) = \deg_{F_1}(v) + \deg_{F_2}(v)$.

$\text{chainend}_{v,F_y}(v') = v'' \in V(G)$ with $\deg_{F_y}(v'') > 2$ and $\exists v_1, \ldots, v_\ell \in V(G)$ such that $\deg_{F_y}(v_i) = 2$ and $\{v', v_1\}, \{v_1, v_2\}, \ldots, \{v_{\ell-1}, v_\ell\}, \{v_\ell, v''\} \in F_y$, for $y \in \{1, 2\}$ and $i \in \{1, \ldots, \ell\}$.

Intuitively, we only consider one forest at a time. The chainend starts when a vertex with degree one gets removed from the forest. We then look if by removing that vertex a new degree one vertex is created. We recursively remove the degree one vertex until the chain stops at a vertex that has a degree higher than one. The index v of the chainend indicates the selection time at which the chain appears. The index F_y indicates in which forest the path is found.

We also need to introduce another term. A vertex v *collapses* in a chainend if it part the chainend chain ($v \in \{v_1, \ldots, v_\ell\} \subseteq V(G)$, where $v_1, \ldots, v_\ell$ refer to the vertices from the chainend definition).

We can now categorize $\deg^*_Q(v)$ more precisely to distinguish between neighbors of v at selection time in Q that are

a) larger than v with respect to $>_{\text{select}}$.

b) smaller than v with respect to $>_{\text{select}}$ and start a chain of collapses when v is removed that ends at a vertex v' that is larger than v with respect to $>_{\text{select}}$.

c) do not fall in one of the two previous categories.

Recall that $X_>(v)$ includes all vertices that are selected in X before v. We define the following. Let $G' = G - X_>(v)$, $G'' = G' - \{v\}$. We assume that both G' and G'' have a minimal degree of three after applying the reduction rules for degree at most two on both of them.

- $\deg^*_{Q,>}(v) := |\{v' \in N^*(v) \cap Q \mid v' >_{\text{select}} v \text{ and } \deg_{G'}(v') > 2\}|$
  (Number of neighbors of v at selection time in Q that are larger than v with respect to $>_{\text{select}}$.)

- $\deg^*_{Q,<}(v) := |\{v' \in N^*(v) \cap Q \mid v >_{\text{select}} v' \text{ and } ((\deg_{F_1}(v') = 2 \text{ and } v'' = \text{chainend}_{v,F_1}(v')) \text{ or } (\deg_{F_2}(v') = 2 \text{ and } v'' = \text{chainend}_{v,F_2}(v'))) \text{ with } v'' >_{\text{select}} v\}|$.

  (Number of neighbors of v at selection time in Q that are smaller than v with respect to $>_{\text{select}}$ and start a chain of collapses in either forest when v is removed that ends at a vertex v'' that is larger than v with respect to $>_{\text{select}}$.)

- $\deg^*_{Q,\text{rest}}(v) := \deg^*_Q(v) - \deg^*_{Q,>}(v) - \deg^*_{Q,<}(v)$

It is obvious that $\deg^*_Q(v) = \deg^*_{Q,>}(v) + \deg^*_{Q,<}(v) + \deg^*_{Q,\text{rest}}(v)$. We define the neighborhoods $N^*_{Q,>}(v)$ and $N^*_{Q,<}(v)$ accordingly. In particular, we can derive the following inequality, which will become handy.

$$\deg^*(v) \geq \deg^*_{Q,>}(v) + \deg^*_{Q,<}(v) \tag{6.1}$$

Now we make the following estimation using Inequality 6.1.

$$\begin{aligned}
|X| &= \sum_{v \in X} \frac{\deg^*(v)}{\deg^*(v)} \geq \sum_{v \in X} \frac{\deg^*_{Q,>}(v) + \deg^*_{Q,<}(v)}{\deg^*(v)} \\
&= \sum_{v \in X} \sum_{u \in Q} \frac{\mathbb{I}_{u \in N^*_{Q,>}(v)} + \mathbb{I}_{u \in N^*_{Q,<}(v)}}{\deg^*(v)} \\
&= \sum_{v \in X} \sum_{u \in Q} \frac{\mathbb{I}_{u \in N^*_{Q,>}(v)} + \mathbb{I}_{\exists u' \in N^*_{Q,<}(v) \text{ s.t. } (u=\text{chainend}_{v,F_1}(u') \text{ or } u=\text{chainend}_{v,F_2}(u'))}}{\deg^*(v)}
\end{aligned}$$

Next, we observe the following:

$$\mathbb{I}_{u \in N^*_{Q,>}(v)} + \mathbb{I}_{\exists u' \in N^*_{Q,<}(v) \text{ s.t. } (u=\text{chainend}_{v,F_1}(u') \text{ or } u=\text{chainend}_{v,F_2}(u'))} > 0 \Rightarrow u >_{\text{select}} v \Rightarrow \deg^*(u) \geq \deg^*(v).$$

This allows us to continue our estimation as follows.

$$\begin{aligned}
|X| &\geq \sum_{v \in X} \sum_{u \in Q} \frac{\mathbb{I}_{u \in N^*_{Q,>}(v)} + \mathbb{I}_{\exists u' \in N^*_{Q,<}(v) \text{ s.t. } (u=\text{chainend}_{v,F_1}(u') \text{ or } u=\text{chainend}_{v,F_2}(u'))}}{\deg^*(u)} \\
&= \sum_{u \in Q} \frac{\sum_{v \in X} \mathbb{I}_{u \in N^*_{Q,>}(v)} + \mathbb{I}_{\exists u' \in N^*_{Q,<}(v) \text{ s.t. } (u=\text{chainend}_{v,F_1}(u') \text{ or } u=\text{chainend}_{v,F_2}(u'))}}{\deg^*(u)}
\end{aligned}$$

Now it is not hard to see that the following inequality holds.

$$\deg^*(u) \geq \sum_{v \in X} \mathbb{I}_{u \in N^*_{Q,>}(v)} + \mathbb{I}_{\exists u' \in N^*_{Q,<}(v) \text{ s.t. } u=\text{chainend}_v(u')} \tag{6.2}$$

However, we can improve this bound by observing that

1. if $\sum_{v \in X} \mathbb{I}_{u \in N^*_{Q,>}(v)} + \mathbb{I}_{\exists u' \in N^*_{Q,<}(v) \text{ s.t. } (u=\text{chainend}_{v,F_1}(u') \text{ or } u=\text{chainend}_{v,F_2}(u'))} > 0$, then u collapses eventually, and

2. in this case at least four neighbors of u at selection time are not counted, namely the two that eventually collapse in F_1 and F_2 and cause u to collapse as well and the two that collapse after u collapsed.

Furthermore, we have that $\deg^*(u) \geq 5$ (see Section 5.1). Hence, we can conclude that the following holds.

$$\deg^*(u) - 4 \geq \sum_{v \in X} \mathbb{I}_{u \in N^*_{Q,>}(v)} + \mathbb{I}_{\exists u' \in N^*_{Q,<}(v) \text{ s.t. } (u=\text{chainend}_{v,F_1}(u') \text{ or } u=\text{chainend}_{v,F_2}(u'))} \tag{6.3}$$

This allows is to continue our estimation as follows.

$$|X| \geq \sum_{u \in Q} \frac{\deg^*(u) - 4}{\deg^*(u)}$$

Now, again using that $\deg^*(u) \geq 5$ we arrive at

$$|X| \geq \frac{1}{5}|Q|. \tag{6.4}$$

Recall that we assumed for contradiction that there is a solution $X \subseteq X^\star \subseteq V$ with $|X^\star| \leq k$ and $X^\star \cap Q = \emptyset$, that is, $X^\star$ contains all vertices of X and no vertex from Q. We know that

$$|X| \overset{(7.4)}{\geq} \frac{1}{5}|Q| \overset{\text{Theorem 6.1}}{>} \frac{1}{5}5k. \tag{6.5}$$

Thus it follows that

$$|X^\star| \geq |X| > k. \tag{6.6}$$

This contradicts our original assumption that there exists a solution of size at most k and the proof is complete. It follows that there cannot exist a valid solution for if our undeletable vertex set size surpasses $5k$. □

6.2 Running Time

We will show in this section that Arboricity-two-reduction is fixed-parameter tracable. To calculate the running time for the entire algorithm, we first have to calculate the running time for the two subroutines of the base case (see Algorithm 3 and Algorithm 4), since they can get called in each recursive step.

Theorem 6.2. *It can be decided in $O(n^2)$ time whether a 3-4-graph G can be reduced to arboricity two by deleting at most k vertices.*

Proof. Algorithm 3 consists of two consecutive steps. First the algorithm has to find every maximum 2-edge-connected-component. This has been proven to be possible in $O(n + m)$ time [7]. Since we consider graphs with a maximum degree of four, the amount of edges is at most twice the amount of vertices, resulting in a running time of $O(n)$. After that, we iterate over the 2-edge-connected-components $O(n)$ times and

check for each one if it is problematic. Recall that a graph is problematic if it has a set density of three. Hence, checking whether a graph is problematic can be done in $O(n)$ time. Removing a vertex from a problematic graph can be done in linear time. Putting those steps together we arrive at a running time of $O(n^2)$ for Algorithm 3. □

Theorem 6.3. *The running time of Algorithm 4 is in $n^2 \cdot k^{O(k^4)}$ time.*

Proof. The guessing algorithm (see Algorithm 4) consists of two steps. The first step initializes Z and the second step tries out each possible solution. Recall that F is $V(G) \setminus Q$. We initialize Z by iterating over all vertices in $G[F]$ and adding all vertices with degree three or four, for which all their edges are connected to a vertex in $G[Q]$. The iteration is done in $O(n)$ time and the checks are done in linear time, yielding $O(n)$ for the first part of the algorithm. For the second part we have to calculate how many possible combinations there are to choose a solution. There are at most $(\binom{|Q|}{4} + \binom{|Q|}{3})$ categories for picking either three vertices or four vertices out of Q. Since Q has size at most k this means there are at most $(\binom{5k}{4} + \binom{5k}{3})$ categories. We know for a category C that we can delete at most $\max\{k, |C|\}$ vertices. Computing the arboricity after every combination takes $O(n^2)$ time. This yields a running time of $n^2 \cdot O(k^{(\binom{5k}{4}+\binom{5k}{3})})$, or $n^2 \cdot k^{O(k^4)}$. □

We now have all necessary components to calculate the running time of the entire algorithm (see Algorithm 2).

Theorem 6.4. *It can be decided in $n^2 \cdot k^{O(k^4)}$ time whether a graph G can be reduced to arboricity two by deleting at most k vertices.*

Proof. To analyze the total running time, we will show the running time for each step of Algorithm 2 and then put everything together. In the first two steps the algorithm checks for simple return conditions in linear time. The algorithm then applies the reduction rules. There are n iterations, in which every vertex gets processed in linear time. Note that recursively calling Algorithm 2 in line 5 does not increase our running time here since it is only called a linear amount of times. Thus the running time of the FOR-loop yields $O(n)$ time. Next in line 7 the algorithm picks the vertex with the highest degree. A naïve algorithm to find the highest degree vertex iterates over every vertex in the graph, resulting in $O(n)$. We then call Algorithm 3 in line 9 of Algorithm 2, which we proved in Theorem 6.2 to be computable in $O(n^2)$ time. Finally, in line 13 we check the arboricity. As mentioned earlier, this is possible in $O(m^2)$ time [9]. We can use a small trick to prove that our algorithm can check the arboricity in $O(n^2)$ time. The trick works as follows. We first calculate the set density in linear time. If it is three, then the arboricity is three as well and we are done. If it is two, then we know that $m < 2n$ because for $m \geq 2n$ the set density is at least three. Therefore we can bound m in $O(n)$ and we obtain a running time of $O(n^2)$ to check the arboricity. Hence, putting everything together, the processing of each vertex in the base step can be done in $O(n^2)$ time.

We proved in Theorem 6.3 that Algorithm 4 has a running time of $n^2 \cdot k^{O(k^4)}$. Thus the total running time for processing each vertex is $n^2 \cdot k^{O(k^4)}$.

The next task is to calculate how many recursive calls Algorithm 2 can do. Recall that we proved in Section 6.1, that finding a solution with Algorithm 2 implies that $|X|$ is at most of size k and $|Q|$ is at most $5k$. Hence for every solution there must exist an execution path of length at most $6k$. Otherwise, all execution paths not leading to a solution also have a length of at most $6k + 1$, since we return "NO" if a solution has not been found after $5k + 1$ steps or $6k$ steps, respectively. Thus the search tree has a depth of $6k + 1$, resulting in $O(2^{6k})$ recursive calls.

Combining everything we have a total running time of

$$n^2 \cdot O(2^{6k}) \cdot k^{O(k^4)} = n^2 \cdot k^{O(k^4)}.$$

This concludes the proof of Theorem 6.4. □

This proves that Arboricity-two-reduction is fixed-parameter tractable with parameter k.

7 Conclusion

In our work, we adapted an existing algorithm idea for solving the Feedback Vertex Set to solve Arboricity-two-reduction. The Feedback Vertex Set algorithm [1] used a greedy-branching method, where the vertex with the maximum degree is selected and recursively checked if a solution with the vertex in the solution set exists, otherwise the algorithm would put it in a set of undeletable vertices, which cannot be contained in the solution. The main difference to the Feedback Vertex Set was the base case, which terminates the recursion. For Arboricity-two-reduction to work in FPT time, we had to figure out an FPT-time base case algorithm to process maximum-degree-four graphs. In comparison, the Feedback Vertex Set algorithm solved the base case for maximum-degree-two graphs, by simply deleting any vertex of every remaining cycle. The base case for Arboricity-two-reduction was significantly more challenging. We split up the base case into two subroutines. For our first subroutine approach was to divide the remaining graph into subgraphs, which we then process individually. This subroutine solves Arboricity-two-reduction for maximum-degree-four graphs in polynomial time. This means that we can also use this subroutine independently as an algorithm for input graphs with maximum-degree four and compute them in polynomial time. Then the second subroutine processes the vertices that connect the remaining graph to the undeletable vertex set Q with an edge. By changing the base case to an earlier termination condition, another main running time property changed. Our FPT-time base case allowed us to prove that we do not have to recurse over every possible solution, but can terminate the recursion as soon as either of our sets becomes too large. This discovery allowed us to achieve fixed-parameter tractability for Arboricity-two-reduction.

7.1 Future Work

The main discovery of this thesis was the extended functionality of the greedy-branching method, adapted from Feedback Vertex Set to Arboricity-two-reduction. An interesting question for future work is to find further problems which could use the method of branching over a solution set and an undeletable vertex set for vertex-deletion problems or even an undeletable edge set for edge-deletion problems. We were able to utilize this method to solve Arboricity-two-reduction. Obviously the question arises, whether it is possible to solve the general case arboricity-p-reduction for any $p \in \mathbb{N}$. The core problem for this task is that the base cases would need to be extended to higher-degree graphs gradually with p for the algorithm to stay in FPT-time. It would also be very interesting to check how fast algorithms can solve Feedback Vertex Set

and Arboricity-two-reduction for actual data sets, especially very big ones and to compare the algorithm to other existing methods [3, 11]. The algorithm for Feedback Vertex Set by Kociumaka and Pilipczuk [11] is currently the fastest one we know of. To the best of our knowledge, no implementations for Arboricity-two-reduction exist to this day, so this question remains for the future.

Literature

[1] Y. Cao. "A Naive Algorithm for Feedback Vertex Set". In: *Proceedings of the 1st Symposium on Simplicity in Algorithms, (SOSA '18)*. Vol. 61. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 1:1–1:9 (cit. on pp. 5, 7, 13, 17, 19, 23, 47).

[2] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015 (cit. on p. 15).

[3] H. Dell, T. Husfeldt, B. M. P. Jansen, P. Kaski, C. Komusiewicz, and F. A. Rosamond. "The First Parameterized Algorithms and Computational Experiments Challenge". In: *Proceedings of the 11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*. Vol. 63. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 30:1–30:9 (cit. on p. 48).

[4] R. Diestel. *Graph Theory, 5th Edition*. Vol. 173. Graduate Texts in Mathematics. Springer, 2016 (cit. on p. 15).

[5] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013 (cit. on p. 15).

[6] D. Eppstein, M. Löffler, and D. Strash. "Listing All Maximal Cliques in Large Sparse Real-World Graphs". In: *ACM Journal of Experimental Algorithmics* 18 (2013), 3.1:3.1–3.1:3.21 (cit. on p. 12).

[7] K. P. Eswaran and R. E. Tarjan. "Augmentation problems". In: *SIAM Journal on Computing* 5.4 (1976), pp. 653–665 (cit. on pp. 34, 44).

[8] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006 (cit. on p. 15).

[9] H. N. Gabow and H. H. Westermann. "Forests, frames, and games: algorithms for matroid sums and applications". In: *Algorithmica* 7.1-6 (1992), p. 465 (cit. on pp. 26, 45).

[10] P. A. Golovach and Y. Villanger. "Parameterized complexity for domination problems on degenerate graphs". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 2008, pp. 195–205 (cit. on p. 12).

[11] T. Kociumaka and M. Pilipczuk. "Faster deterministic feedback vertex set". In: *Information Processing Letters* 114.10 (2014), pp. 556–560 (cit. on p. 48).

[12] J. M. Lewis and M. Yannakakis. "The node-deletion problem for hereditary properties is NP-complete". In: *Journal of Computer and System Sciences* 20.2 (1980), pp. 219–230 (cit. on p. 11).

[13] D. R. Lick and A. T. White. "k-Degenerate graphs". In: *Canadian Journal of Mathematics* 22.5 (1970), pp. 1082–1096 (cit. on p. 12).

[14] J. Luo, H. Molter, and O. Suchý. "A Parameterized Complexity View on Collapsing k-Cores". In: *Proceedings of the 13th International Symposium on Parameterized and Exact Computation (IPEC '18)*. Vol. 115. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 7:1–7:14 (cit. on p. 13).

[15] L. Mathieson. "The parameterized complexity of editing graphs for bounded degeneracy". In: *Theoretical Computer Science* 411.34-36 (2010), pp. 3181–3187 (cit. on pp. 12, 13).

[16] D. W. Matula and L. L. Beck. "Smallest-last Ordering and Clustering and Graph Coloring Algorithms". In: *Journal of the ACM* 30.3 (1983), pp. 417–427 (cit. on p. 12).

[17] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006 (cit. on p. 15).

[18] G. Philip, V. Raman, and S. Sikdar. "Polynomial Kernels for Dominating Set in Graphs of Bounded Degeneracy and Beyond". In: *ACM Transactions on Algorithms (TALG)* 9.1 (2012), 11:1–11:23 (cit. on p. 12).