Universiteit Utrecht

Faculteit Bètawetenschappen

Parameterized Algorithms in a Streaming Setting

Master Thesis

Jelle Oostveen
ICA-5707757

Computing Science

Supervisors:

Dr. Erik Jan van Leeuwen
Department of Information and Computing Sciences

Prof. Dr. Hans L. Bodlaender
Department of Information and Computing Sciences

July 15, 2020

Abstract

Saving large graphs in memory can be problematic. Streaming the graph provides a solution, where the graph arrives as a stream of edges. Different streaming models can be used to assume a variable amount of information present in the stream. In combination with parameterized complexity, we can find algorithms using sublinear memory, where for non-parameterized algorithms it can be proven that we require linear memory. In this work, we investigate a broad range of problems in the streaming setting, including varying approaches to solve them. Different approaches can lead to an interesting trade-off in the number of times we inspect the stream and the memory usage. Hence, we explore both kernelization and direct algorithmic approaches for solving problems such as Π-free Deletion, Π-free Editing, Vertex Cover, and Edge Dominating Set. In terms of parameterization, we focus on using the parameter *vertex cover*, but we will also see *solution size* and *tree-depth* being used. We will see a range of upper bound results, which are partially direct adaptations of non-streaming algorithms, and partially new work. We will also see some lower bounds on the memory use given the number of passes we make over the stream.

Contents

1 Introduction

In modern day systems, increasingly large networks occur. We can consider examples of companies like Facebook or Twitter having billions of users [27, 62], creating a network with billions of nodes and connections. On such networks fast algorithms are essential for analysis and queries. However, time efficiency is not the only concern with such large networks. More specifically, if we want to run algorithms on large graphs we usually assume that the entire structure is in memory. Sadly enough, this is not always possible due to large size, which motivates other methods of executing algorithms on graph structures.

One of the main methods for doing this is using a graph stream. That is, the graph arrives in memory as a stream of edges which we can read one edge at the time. This has the added benefit that we can also look at dynamic streams, where edge deletions might occur in the stream. In both cases we look to develop space-efficient algorithms that read the graph in one or more passes of the stream. It is interesting to note that, although we are looking for space-efficient algorithms, there is still a trade-off in the complexity. That is, it is not unusual to have some algorithm be space optimal but use an exponential number of passes of the stream, while another algorithm perhaps uses more space but drastically fewer passes of the stream.

Considering the great success of parameterized complexity in time-efficient algorithms, it is a natural analogy to try and develop parameterized space-efficient algorithms that work on a graph stream. The study in this field was initiated by Fafianie and Kratsch [28] and Chitnis *et al.* [20] and proved some initial success. The main body of results consists of both algorithmic kernels and direct FPT algorithms, usually parameterized by the solution size k. These results vary greatly in space-efficiency and the number of passes used, which leaves room for an interesting analysis on the trade-off of these two factors, where improvement on one, or both, can be obtained. We can also wonder whether techniques from classic FPT algorithms can be used in the streaming setting, e.g. branching algorithms. Combining this with the fact that different parameters than the solution size are fairly unexplored in the field, we are left with a lot of room for new discoveries.

In this master thesis, the main aim is to continue study in this field, focussing on both the trade-off between memory use and the number of passes, as well as using the vertex cover size as a parameter. We will first give a small overview of preliminary knowledge in Section 2, followed by an overview of the literature in the field in Section 3. Our analysis starts with Π-free vertex deletion problems in Section 4. Following up vertex deletion problems, Section 5 discusses Π-free edge editing problems. We then look at vertex cover in Section 6, and discuss the use of a different streaming model in Section 7. Finally, Section 8 contains concluding remarks and options for future work.

2 Preliminaries and general remarks

2.1 Graph notation

A graph $G = (V, E)$ consists of vertices $v \in V$ and edges $e \in E$. An edge can also be denoted by the vertices it spans, when $u, v \in V$ then $uv \in E$. In this entire thesis, we work on undirected graphs. We denote n as the number of vertices in G, so $|V| = n$, and similarly, $|E| = m$, the number of edges in G. If it is not clear from context what graph we consider, we use $V(G), E(G)$ for respectively the vertices and edges of G. For a set vertices $V' \subseteq V$, denote the subgraph induced by V' as $G[V']$. An induced subgraph consists of the vertices V' and all edges between the vertices of V' present in G. Denote the neighbourhood of a vertex v with $N(v)$ and for a set S denote $N(S)$ as $\bigcup_{v \in S} N(v)$. If we wish to denote v together with its neighbourhood, we write $N[v]$, where $N[v] = N(v) \cup \{v\}$. We use standard asymptotic notation of big-O, $\mathcal{O}$, and if we wish to hide logarithmic factors in n we use $\widetilde{\mathcal{O}}$. Similarly for lower bounds, we use Ω and $\widetilde{\Omega}$. Memory upper or lower bounds will be given in bits, that is, they usually include a factor $\log n$ to accommodate for numbers up to size n requiring $\mathcal{O}(\log n)$ bits to save in memory.

2.2 Parameterized Complexity

Let us give a short overview of notions in parameterized complexity. In parameterized complexity, we are given some parameter as part of the problem input. The idea is that this parameter is small, and so, we can think of algorithms that are expensive in the parameter, but efficient in the rest of the input, giving an overall fast algorithm. Let us denote the problem parameter with k. We call a problem *Fixed Parameter Tractable* (FPT) if there exists an algorithm with a running time of $\mathcal{O}(f(k) \cdot n^{\mathcal{O}(1)})$ where f is some function in k. Another approach is to preprocess the input such that its size becomes bounded by a function in k, while it is still a YES-instance if and only if the original instance was a YES-instance. If this preprocessing can be done in polynomial time, then the remaining instance is called a *kernel*, and obtaining the instance is called *kernelization*. We can then execute any algorithm on the kernel to obtain a solution. The notions of a kernel and FPT are equivalent, that is, a problem is FPT if and only if it has a polynomial time kernelization.

The most standard parameter is the solution size. However, in this thesis we will also see another parameter often present, the vertex cover number. The *vertex cover number* is the size of a minimum vertex cover in the graph. A vertex cover is a set of vertices such that all edges in the graph are incident to some vertex in the vertex cover, see also Section 6 for a formal definition of the Vertex Cover problem. Ideally, the vertex cover number is as small as possible. If both these standard parameters are present, we will usually denote the vertex cover size with K, and the solution size with ℓ, to avoid confusion. We will denote the parameters of a problem in [...] brackets, so a problem A parameterized by vertex cover number and solution size is denoted by A [VC, ℓ].

In Section 7, we will also work with the parameter *tree-depth*. To understand what tree-depth is, we first need to define a *tree-depth decomposition*.

Definition 2.1. A *tree-depth decomposition* of a graph G is a rooted forest T with the same set of vertices as G, that is, $V(T) = V(G)$. The following property must also hold: for every pair of vertices $v, w \in V(G)$ that have $vw \in E(G)$ it must be that v and w have an ancestor-descendent relationship in T. The *depth* of T is the maximum height of a tree in T. The *tree-depth* of a graph G is now given as the minimum depth over all tree-depth decompositions of G.

For illustration, let us go over some small examples. A graph without edges has tree-depth 1, as we can make a tree-depth decomposition where every tree is a single vertex. A complete clique of n vertices has tree-depth n, as all vertices must have an ancestor-descendent relationship in any tree-depth decomposition.

To relate tree-depth to other parameters, we give the definitions of *treewidth* and *pathwidth*, taken from [56].

Definition 2.2. A *tree decomposition* of an undirected graph G is a tree T together with a collection of sets of vertices of G (called *bags*) X_t indexed by nodes $t \in T$, such that:

- every vertex of G is in at least one bag,
- for every edge uv of G, there is a bag containing both u and v, and
- for every vertex v of G, the set $\{t \in T \mid v \in X_t\}$ induces a connected subtree of T.

The width of a tree decomposition is defined as $\max_{t \in T} |X_t| - 1$. The *treewidth* of G is the minimum width over all possible tree decompositions of G.

Definition 2.3. A *path decomposition* of an undirected graph G is a tree decomposition $(T, (X_t)_{t \in T})$ in which T is a path. The *pathwidth* of G is the minimum width over all possible path decompositions of G.

It is known that we can construct a path decomposition with pathwidth equal to the tree-depth from any tree-depth decomposition [56]. Therefore, the pathwidth of a graph G is always equal to or smaller than the tree-depth of G. As the path decomposition of a graph is a special case of a tree decomposition of a graph, it then also follows that the treewidth of a graph G is always equal to or smaller than the tree-depth of G [56].

2.3 Streaming

In this thesis, we work on the streaming model. This is a model where the input graph G is given as a stream of edges. A *pass* over the stream means we view every edge in the stream once, and so, after a pass we have seen the entire graph. There are four different streaming models we study, with a varying degree of information. These four models are those usually studied in the literature, for example, see [8, 21, 45, 46]. The *Edge Arrival* (EA) streaming model allows the edges to appear in any order in the stream. The *Vertex Arrival* (VA) streaming model requires the edges to appear per vertex, that is, if we have seen the vertices $V' \subseteq V$ already, and the next vertex is w, then the stream contains the edges between w and the vertices in V' present in the graph. The *Adjacency List* (AL) streaming model gives us the most information, as this model requires the edges to arrive per vertex, but also for every vertex we see every adjacency immediately. This means we effectively see every edge twice in a single pass, once for both of its vertices. The last model is the *Dynamic Edge Arrival* (DEA) streaming model, which is an EA stream that can also contain edge removals; hence, it is dynamic.

The aim of using the streaming model is to develop memory-efficient algorithms, as $\widetilde{\mathcal{O}}(m) = \widetilde{\mathcal{O}}(n^2)$ bits of memory is too large in our use case (we do not want to have the entire graph in memory, hence we use a streaming model). Therefore, algorithms working in the streaming model should have both the number of passes used and the amount of memory used listed. Optionally, computation time between passes/for each step can also be listed, but as we focus less on this aspect, we allow unbounded computation and omit running times. However, the number of passes can still be viewed as some factor of running time, as passing over a (large) stream might require quite some computation time. Given all these focus areas, we will often consider a pass/memory trade-off, where we can opt for one algorithm using non-optimal memory but very few passes or another using optimal memory but very many passes. This also means that there might be value in exploring another approach to a solved problem, which could provide with an algorithm on the other side of the pass/memory trade-off. An example of this is the Vertex Cover $[k]$ problem, where a kernel exists that can be found in one pass and $\mathcal{O}(k^2 \log n)$ bits of memory [18], while on the other hand an algorithm using $\mathcal{O}(k \log n)$ bits of memory and 2^k passes exists [17]. A kernel found by a streaming algorithm we will call a *streaming kernel*.

Using the combination of parameterized complexity and streaming is well suited, as many problems in the streaming setting have memory lower bounds of $\Omega(n)$ or $\Omega(n \log n)$ bits [29, 30]. The use of a parameter avoids these lower bounds, as we develop algorithm specifically suited for the scenarios where some aspect of the input is small. This should lead to much more memory-efficient algorithms. Our goal in this regard is to find algorithms using $\mathcal{O}(f(k) \cdot \log n)$ bits of memory, where f is some function dependent on k. The $\log n$ factor in the memory can hardly be avoided, as representing sets of vertices/edges/elements requires $\log n$ bits per element. However, the function $f(k)$ can optimally be very small, and when the parameter k is small, this creates very memory-efficient algorithms.

2.4 On the memory complexity of branching algorithms

With the streaming model having a focus on efficient memory use, it is useful to do some analysis on the memory complexity of a standard algorithmic type of algorithm, branching algorithms. Let us assume we have some branching algorithm $\mathcal{A}$ that branches on b options at most k times, leading to a search tree of size $\mathcal{O}(b^k)$. Let us also assume that in each branching node, the algorithm requires $\widetilde{\mathcal{O}}(x)$ bits of memory to save the current branching solution and perhaps other factors for computation (so x could be k). The worry is that the branching procedure might only require $\widetilde{\mathcal{O}}(x)$ bits in each step, but that the total overhead with all the branching steps leads to far greater memory use. If we naively copy sets when we branch (so each branch gets $S \cup \{v\}$ where v ranges over b different vertices or edges), we can get a memory complexity of at least $\widetilde{\Omega}(xk)$ bits because each of these sets is in memory.

However, we can be more diligent with our memory usage. Notice that in a branching step, the memory needed only differs in one point: the last vertex or edge added. Therefore, instead of copying the set to some new space, the branches can use the already existing set S in memory, and only add their own adjustment to it. This makes the memory use already much more efficient. However, we also need to remember on what we branch, so that when we return out of recursion, we can correctly continue branching. Seeing as we branch on b options and the search tree has depth at most k, this can lead to a $\widetilde{\mathcal{O}}(bk)$ memory usage. However, if b is a constant in the algorithm, this memory complexity will probably be negligible in comparison to $\widetilde{\mathcal{O}}(x)$. An alternative if b is not a constant is to recompute the b options every time we return out of recursion, which gives some extra b factor in the number of passes. This analysis does give us that, for the worst case memory complexity of a branching algorithm, we only have to consider the worst case memory use of a single branching step and with b and k we can conclude the rest of the complexity. So, in future, when we discuss some branching algorithm as described above, showing that the worst case memory use is at most $\widetilde{\mathcal{O}}(x)$ in a branch gets us that the entire algorithm uses $\widetilde{\mathcal{O}}(x + bk)$ bits of space, which will commonly be $\widetilde{\mathcal{O}}(x)$ bits of space when b is a constant and $k = \mathcal{O}(x)$. If we opt to use extra passes to avoid having the $\widetilde{\mathcal{O}}(bk)$ memory complexity, we will mention this explicitly.

3 Previous Work on Parameterized Streaming Algorithms

The study of solving problems on streams has its origins in algorithms that count or estimate some statistics over large amounts of data [32, 49, 60]. In these studies, the stream does not always describe a graph, but is usually some sequence of numbers, elements, or events. Well-known works include that of Alon *et al.* [4] and Henzinger *et al.* [37], results of which include approximating frequency moments over a stream of elements, and the computation of various degree-queries over graph streams. A survey on graph streams can be found in [45].

The study of parameterized algorithms in a streaming setting was initiated by Fafianie and Kratsch [28] and Chitnis *et al.* [20].

In 2014, Fafianie and Kratsch [28] consider the Edge Arrival model, and show kernalizations and lower bounds. One-pass kernels are given for both d-Hitting Set $[k]$ and d-Set Matching $[k]$ parameterized on the solution size k. These kernels are closely related to the Sunflower Lemma. In contrast, memory lower bounds of $|E| - \mathcal{O}(1)$ bits are given for a large list of problems if only one pass is allowed, including e.g. Edge Dominating Set $[k]$ and Cluster Vertex

Deletion [k]. Intuitively, these lower bounds follow from a pigeonhole principle, where if we use 'too little' memory, there must be YES-instances we cannot distinguish from NO-instances. However, if we allow two passes, Edge Dominating Set [k] admits a $\mathcal{O}(k^3 \log n)$ size kernel. This kernel makes use of a $2k$ vertex cover kernel finding algorithm as its first pass, after which the second pass is used to 'remember' edges that might be very useful in finding the optimal solution.

In 2015, Chitnis *et al.* [20] provide results for finding maximal matchings and vertex covers. For the scenario where the stream contains only edge insertions, Chitnis *et al.* provide a space-optimal algorithm for Vertex Cover [k] using $\widetilde{\mathcal{O}}(k^2)$ space and one pass, and show that this is space-optimal for one pass. For dynamic streams, under the promise that at any time there exists a vertex cover of size at most k, single pass, $\widetilde{\mathcal{O}}(k^2)$ space algorithms are given for Vertex Cover [k] and Maximal Matching [k]. This promise can be removed, but this worsens the results significantly. To achieve all this, sketching structures are used in combination with ℓ_0-samplers, which makes the algorithm randomized. These sketches attempt to store the essential information of the graph stream without using too much space, which keeps the memory of the algorithms below $\widetilde{\mathcal{O}}(k^2)$ bits.

In 2016, Chitnis *et al.* [18] remove the need for the above promise and provide space efficient kernels of size $\widetilde{\mathcal{O}}(k^2)$ for Maximal Matching [k] and Vertex Cover [k] in the dynamic stream setting. To achieve this, they make use of a sampling method which is biased towards maintaining the structure of the graph instead of being biased on high degree vertices. A $\widetilde{\mathcal{O}}(k^d)$ space kernel is also given for d-Hitting Set [k], which is a generalization of the result for Vertex Cover [k]. They also show that single pass algorithms with an insert-only stream for d-Hitting Set [k] require at least $\Omega(k^d)$ space and for Edge Dominating Set [k] at least $\Omega(n)$ space. Furthermore, they provide many general lower bounds showing that any p-pass algorithm requires at least $\Omega(n/p)$ space, including e.g. Triangle Packing [k] and Cluster Vertex Deletion [k].

In 2019, Bishnu *et al.* [8] provided a wide variety of results best described in a table, see Table 1. Let us highlight some of these results. The first thing to note is the use of four different streaming models: Edge Arrival, Dynamic Edge Arrival, Vertex Arrival, and Adjacency List. The main newly introduced component to the area is the use of other parameters than the solution size ℓ, namely, the vertex cover size K, which proves quite potent. This parameter in combination with the Adjacency List model gives rise to new algorithms. The algorithm for Cluster Vertex Deletion makes use of the same sampling primitive as in [18]. The algorithm for the general $\mathcal{F}$-Subgraph Deletion [VC] makes use of a structural subroutine called *Common Neighbour*, which is a maximal matching together with sufficient common neighbours of these matched vertices. This subroutine then leads to the result for $\mathcal{F}$-Subgraph Deletion [VC], and so also for Triangle Deletion [VC], Feedback Vertex Set [VC], and both Even Cycle Transversal [VC] and Odd Cycle Transversal [VC].

Also in 2019, Chitnis and Cormode [17] formalize the complexity classes of space-efficient parameterized streaming algorithms, see Figure 1. The idea of these complexity classes is their focus on memory use. The outer complexity class, *BrutePS*, allows us to use $\mathcal{O}(n^2)$ space, and so save the entire graph into memory. Formally, a problem is in the class BrutePS if it can be solved using $\mathcal{O}(n^2)$ bits[1]. This is the outer limit, as streaming the input graph is not useful in this class. The inner-most complexity class, *Fixed-Parameter Streaming (FPS)*, is named after its similarity with FPT, as a problem is in FPS if it can be solved using $\widetilde{\mathcal{O}}(f(k))$ bits, where $f(k)$ is some function in k. Without the use of parameterized complexity, most problems would

[1]As remarked by Chitnis and Cormode [17], formally, we would actually need to consider the following 7-tuple when deciding the complexity class of a problem: [Problem, Parameter, Space, Number of Passes, Type of Algorithm, Approx. Ratio, Type of Stream].

Problem	Parameter	Upper Bound (algorithm)	Lower Bound (hardness)
$\mathcal{F}$-Subgraph Deletion	ℓ		$(\mathcal{M}, n\log n, 1)$-hard, $(\mathcal{M}, n/p, p)$-hard*
	K	$(\text{AL}, \Delta(\mathcal{F}) \cdot K^{\Delta(\mathcal{F})+1}, 1)$-str	$(\text{VA/EA/DEA}, n/p, p)$-hard
$\mathcal{F}$-Minor Deletion	ℓ		$(\mathcal{M}, n\log n, 1)$-hard, $(\mathcal{M}, n/p, p)$-hard
	K	$(\text{AL}, \Delta(\mathcal{F}) \cdot K^{\Delta(\mathcal{F})+1}, 1)$-str	$(\text{VA/EA/DEA}, n/p, p)$-hard
Feedback Vertex Set, Even Cycle Transversal, Odd Cycle Transversal	ℓ		$(\mathcal{M}, n\log n, 1)$-hard, $(\mathcal{M}, n/p, p)$-hard*
	K	$(\text{AL}, K^3, 1)$-str	$(\text{VA/EA/DEA}, n/p, p)$-hard
Triangle Deletion	ℓ		$(\text{VA/EA/DEA}, n\log n, 1)$-hard, $(\text{VA/EA/DEA}, n/p, p)$-hard*
	K	$(\text{AL}, K^3, 1)$-str	$(\text{VA/EA/DEA}, n/p, p)$-hard
Cluster Vertex Deletion	ℓ		$(\text{VA/EA/DEA}, n/p, p)$-hard*
	K	$(\mathcal{M}, K^2\log^4 n, 1)$-str	

Table 1: Table overviewing results from Bishnu et al. [8]. Results marked with a * is where the EA model result is from [18]. A problem is (M, X, Y)-str when it admits an algorithm in model M using Y passes and $\widetilde{\mathcal{O}}(X)$ bits of memory. A problem is (M, X, Y)-hard when any algorithm in model M that uses Y passes requires at least X bits of memory. Here the parameter ℓ is the solution size and K is the vertex cover number. $\mathcal{M}$ is the set of models AL, EA, VA, DEA. $\Delta(\mathcal{F})$ is the maximum degree over all vertices in all graphs in $\mathcal{F}$.

be stuck in the classes *SupPS* and *SemiPS*, which allow $\widetilde{\mathcal{O}}(f(k) \cdot n^{1+\epsilon})$ and $\widetilde{\mathcal{O}}(f(k) \cdot n)$ bits respectively, where $1 > \epsilon > 0$ and $f(k)$ is some function in k. Parameterized complexity allows us to break through to the classes *SubPS* and FPS, where a problem is in SubPS if it can be solved using $\widetilde{\mathcal{O}}(f(k) \cdot n^{1-\epsilon})$ bits of memory, where $1 > \epsilon > 0$ and $f(k)$ is some function in k.

The algorithmic results given by Chitnis and Cormode are summarized in Table 2. Particularly interesting is the streaming algorithm for Vertex Cover $[k]$ using $\mathcal{O}(k)$ words memory and 2^k passes, as it is a careful adaptation of the regular branching FPT algorithm for Vertex Cover $[k]$ into the streaming setting. Also interesting (although obsolete in efficiency) is the streaming algorithm for Vertex Cover $[k]$ using $\mathcal{O}(k)$ words memory and 2^{2k} passes, which uses iterative compression, a technique that had not yet been used in the streaming setting. Both these results are potentially very useful in discovering new streaming algorithms on the basis of existing branching algorithms.

A recent paper by Agrawal *et al.* [3] discusses more set-oriented problems like the Min-Ones d-SAT $[k]$ problem, where we assume the input arrives in the stream per clause. An algorithmic trade-off is shown where there is the possibility for a streaming algorithm for Min-Ones d-SAT $[k]$ using $\mathcal{O}((k^d + d^{\mathcal{O}(k)})k\log n)$ space and $k+1$ passes, or $\mathcal{O}(k)$ space and $\mathcal{O}(d^k)$ passes. These two results also imply similar results for the Bounded IP problem. The first algorithm in this trade-off makes use of a kernelization for d-Hitting Set $[k]$ to find a set of minimal satisfying assignments. The alternative algorithm is a careful adaption of a branching algorithm for Min-Ones d-SAT $[k]$, which makes for the large number of passes. In the specific case where $d = 2$, we can obtain an algorithm using $\mathcal{O}(k^6)$ space and $k+2$ passes for Min-Ones d-SAT $[k]$, by simplifying the previous general result for this specific case. Complementing the positive results, different lower bounds are given for Min-Ones d-SAT $[k]$ depending on the number of passes,

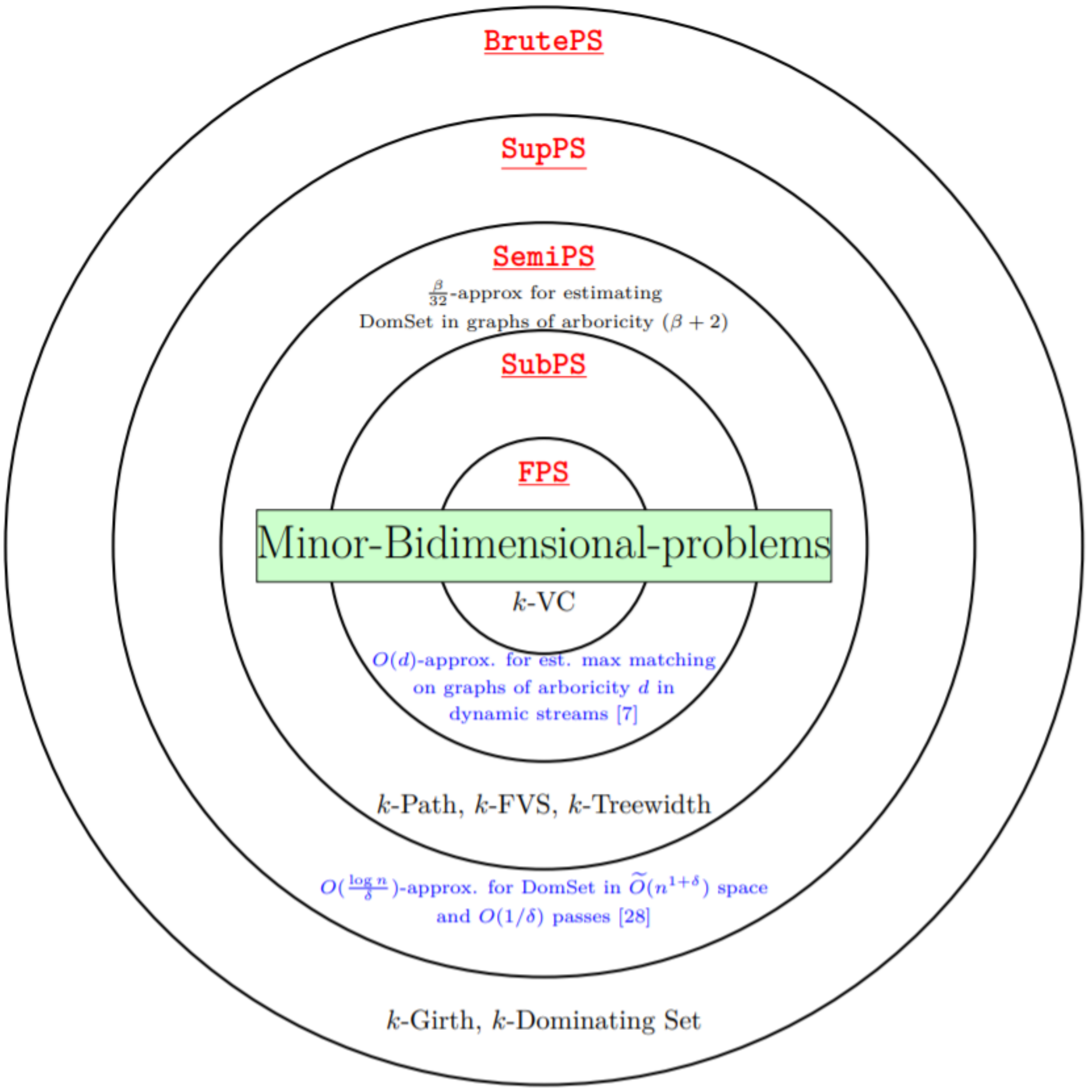
BrutePS
SupPS
SemiPS
$\frac{\beta}{32}$-approx for estimating DomSet in graphs of arboricity $(\beta + 2)$
SubPS
FPS
Minor-Bidimensional-problems
k-VC
$O(d)$-approx. for est. max matching on graphs of arboricity d in dynamic streams [7]
k-Path, k-FVS, k-Treewidth
$O(\frac{\log n}{\delta})$-approx. for DomSet in $\widetilde{O}(n^{1+\delta})$ space and $O(1/\delta)$ passes [28]
k-Girth, k-Dominating Set

Figure 1: A visual overview of the streaming complexity classes for one pass algorithms on EA streams as given by Chitnis and Cormode [17].

Problem	**Passes**	**Model**	**Upper Bound**	**Lower Bound**
$g(r)$-minor-bidimensional problems	1	DEA	$\widetilde{\mathcal{O}}((g^{-1}(k+1))^{10}n)$ words	
Vertex Cover(k)	$2^{2k} \cdot k$	EA	$\mathcal{O}(k)$ words	$\Omega(k)$ words
Vertex Cover(k)	2^k	EA	$\mathcal{O}(k)$ words	$\Omega(k)$ words
FVS(k), Path(k), Treewidth(k)	1	EA	$\mathcal{O}(k \cdot n)$ words	No $f(k) \cdot n^{1-\epsilon} \log^{\mathcal{O}(1)} n$ bits algorithm
FVS(k), Path(k), Treewidth(k)	1	DEA	$\widetilde{\mathcal{O}}(k \cdot n)$ words	No $f(k) \cdot n^{1-\epsilon} \log^{\mathcal{O}(1)} n$ bits algorithm
Girth(k), Dominating Set(k)	1	DEA	$\mathcal{O}(n^2)$ bits	No $f(k) \cdot n^{2-\epsilon} \log^{\mathcal{O}(1)} n$ bits algorithm
$\frac{\beta}{32}$-approximation for Dominating Set(k) on graphs of arboricity β	1	EA	$\widetilde{\mathcal{O}}(n\beta)$ bits	No $f(\beta) \cdot n^{1-\epsilon}$ bits algorithm
AND- and OR-compatible problems	1	EA	$\mathcal{O}(n^2)$ bits	No $\widetilde{\mathcal{O}}(f(k) \cdot n^{1-\epsilon})$ bits algorithm
d-SAT with N variables	1	Clause Arrival	$\widetilde{\mathcal{O}}(d \cdot N^d)$ bits	$\Omega((N/d)^d)$ bits

Table 2: Overview of the results in [17]. The parameter k is the solution size.

including the common lower bound of any p-pass streaming algorithm requiring $\Omega(n/p)$ space.

In almost all of the lower bound proofs in the above papers, extensive use is made of problems from communication complexity. These are problems where two people, Alice and Bob, both have some information and try to find an answer to a certain question by communicating with each other using as few bits as possible. We also distinguish scenarios where we consider one-way communication or two-way communication. For many communication complexity problems lower bounds in the number of bits of communication required are known. This is useful when we reduce a communication complexity problem to a streaming problem where communication is simulated as passing (part of) a stream to the algorithm. From the lower bound for the communication complexity problem then follows a lower bounds in the memory use of such a streaming algorithm. As mentioned, this is how most lower bound proofs are provided in all the above papers. An example of lower bounds proofs using communication complexity can be found in Section 4.6.

4 Π-free Deletion

The focus in this section is on the Π-free Deletion problem. To find streaming algorithms for this problem, we will first look at existing kernels that can be adapted to the streaming setting. These kernels use vertex cover as a parameter, which proves extensively useful. After these kernels we look to create a direct algorithm, with the aim of finding a different pass/memory trade-off. We then continue to a specific case, Odd Cycle Transversal [VC]. Lastly, we will prove some lower bounds for the Π-free Deletion problem.

Vertex cover size as a parameter has already proven to be successful in the streaming setting, as illustrated in [8], which motivates our focus on vertex cover as a parameter.

4.1 Problem Definition and Context

In this section we will work on a fixed family of graphs Π. The problem of Π-free Deletion parameterized by vertex cover can be defined as follows.

Π-free Deletion [VC]
Input: A graph G with a vertex cover X, and an integer $\ell \geq 1$.
Parameter: The size $K := |X|$ of the vertex cover.
Question: Is there a set $S \subseteq V(G)$ of size at most ℓ such that $G - S$ does not contain a graph in Π as an induced subgraph?

By the problem definition, we can see that we want to delete vertices from a graph G such that it is Π-free, that is, no graph in Π occurs as an induced subgraph of G. In regard to the vertex cover as a parameter, it is interesting to note that whenever $\ell \geq K$, we are allowed to delete the entire vertex cover, which removes every edge from the graph. If every graph in Π has at least one edge, then this means we have a trivial solution whenever $\ell \geq K$. Therefore, for the rest of this section, we will assume that every graph in Π has at least one edge (which are interesting cases) and that $\ell \leq K$.

The problem of Π-free Deletion is broad, and many of its specific forms (with some given Π) are well known. Examples include Cluster Vertex Deletion ($\Pi = \{P_3\}$) [40], Odd Cycle Transversal (Π is the set of all graphs containing an odd cycle, see for example [23, Chapter 4.4]), or Vertex Cover ($\Pi = \{K_2\}$, see Section 6).

4.2 Adapting Existing Kernels

The aim of this section is to adapt existing kernels to the streaming model to obtain new results for Π-free Deletion [VC].

4.2.1 Kernel for Characterization by Few Adjacencies

We will first consider kernels given by Jansen in his Phd thesis [42]. In Chapter 4, Jansen provides general kernelization theorems that make extensive use of a single property, that some graph properties can be characterized by few adjacencies. This common property can be used to find kernelizations for general sets of problems, even broader than just Π-free Deletion.

For completeness, we give the formal definition of being characterized by few adjacencies here, as given in [42, Chapter 4].

Definition 4.1. ([42, Chapter 4, Definition 4.1]) A graph property Π is *characterized by* $c_\Pi \in \mathbb{N}$ *adjacencies* if for all graphs $G \in \Pi$, for every vertex $v \in V(G)$, there is a set $D \subseteq V(G) \setminus \{v\}$ of size at most c_Π such that all graphs G' that are obtained from G by adding or removing edges between v and vertices in $V(G) \setminus D$, are also contained in Π.

Jansen shows that graph problems such as Π-free Deletion, parameterized by vertex cover, can be solved efficiently through kernelization when Π is characterized by few adjacencies (and meets some other demands), by making heavy use of the Reduce algorithm he provides. For reference, this algorithm is provided here as Algorithm 1.

Let us shortly describe the essence of why this kernelization is correct. The idea behind the Reduce algorithm is to save *enough* vertices with specific adjacencies in the vertex cover, and those vertices that we forget have equivalent vertices saved to replace them. Which adjacencies to the vertex cover we have to look at can be bounded by making use of the characterization by few adjacencies, as more adjacencies than c_Π are not relevant. What amount of vertices saved is *enough* is another relevant question. In the algorithm this is a variable r, which will come to depend on the solution parameter ℓ and the maximum size of a (vertex-minimal) graph in Π. This way, there are still enough vertices for any occurrence of a graph in Π to appear in the kernel, and to make the vertex deletions as impactful as they would usually be (i.e. if in the original graph we need at least x deletions to remove an occurrence, we still need at least x deletions).

Algorithm 1 Reduce(Graph G, Vertex Cover $X \subseteq V(G)$, $r \in \mathbb{N}$, $c \in \mathbb{N}$) [42, Algorithm 1 on p. 60]

```
for all $Y \in \binom{X}{\leq c}$ and a partition of $Y$ into $Y^+ \cup Y^-$ do
    $Z := \{v \in V(G) \setminus X \mid v$ is adjacent to all of $Y^+$ and none of $Y^-\}$
    Mark $r$ arbitrary vertices of $Z$ (if $|Z| < r$ then mark all of them)
Delete from $G$ all unmarked vertices that are not contained in $X$
```

If we choose the right streaming model, this algorithm can be translated into a streaming setting without much difficulty. As the theorems given by Jansen essentially only make a call to Reduce to achieve their results, we can exchange this algorithm for a streaming equivalent to achieve these theorems in the streaming setting.

Adapting Reduce to the Streaming Model

We will now give an adaptation of the Reduce Algorithm in the streaming model. This adaptation can be found in Algorithm 2, ReduceStr. We state the following theorem on the performance of Algorithm 2.

Theorem 4.2. *For a fixed constant c_Π,* ReduceStr*(G, X, r, c_Π), where G is provided as a stream in the* AL *model, is a one-pass streaming algorithm using $\mathcal{O}((|X| + |X|^{c_\Pi}) \log(n))$ bits of memory resulting in a graph on $\mathcal{O}(|X| + r \cdot |X|^{c_\Pi})$ vertices, output as an* EA *stream.*

Proof. Let us first elaborate on the working of Algorithm 2. The set Z stores all the partitions considered in the original Reduce, together with two counters per partition. The first counter, x, tracks the total amount of vertices already 'marked' with this partition, which may not exceed r. The second counter is reset at every vertex, and tracks whether v is adjacent to the entirety of Y^+. This means these two counters mimic exactly the marking behaviour that Reduce applies, except that the marking is not on arbitrary vertices, but dependent on the order of the stream (this does not impact the correctness). The rest of the algorithm merely interacts with Z correctly and uses some storage, S and V', to make sure the output is constructed correctly without using too much memory. V' remembers which vertices in X we have already seen, to avoid outputting the same edge twice. S saves the set of edges adjacent to a vertex v, and if we mark v, we output S.

Let it be clear by the above motivation that the output of ReduceStr can also be an output of Reduce, and therefore, the algorithm works correctly. Let us analyse the space usage. The main concern is the space usage of the set Z, which contains partitions of sets in $\binom{X}{\leq c}$. There are at most $|X|^c$ such sets, each with at most 2^c partitions, and each set has at most c elements (using $\log n$ space). This means Z uses $c \cdot 2^c \cdot |X|^c \cdot \log n = \mathcal{O}(|X|^c \log n)$ bits of space. The set S is reset at every vertex v (which is not part of the vertex cover), and contains at most all edges incident on v. As v is not an element of the vertex cover, the degree of v is at most $|X|$. So the set S uses at most $\mathcal{O}(|X| \log n)$ bits of space. The set V' has size at most $|X|$, and so uses at most $\mathcal{O}(|X| \log n)$ bits of space. All in all, our memory never exceeds $\mathcal{O}((|X| + |X|^c) \log n)$ bits. □

An interesting observation is that Algorithm 2 outputs the kernel as an EA stream. This should not have a lot of impact, as we probably want to store the entire kernel anyway. However,

Algorithm 2 ReduceStr(Graph $G = (V, E)$ given as a stream in the AL model, Vertex Cover $X \subseteq V(G)$, $r \in \mathbb{N}$, $c \in \mathbb{N}$)

```
1: for each $Y \in \binom{X}{\leq c}$ and a partition of $Y$ into $Y^+ \cup Y^-$ do   ▷ Calculate and store partitions
2:     Store $(Y^+, Y^-, 0, 0)$ in $Z$
3: Store the output vertices $V' \leftarrow \emptyset$   ▷ Required for neatly outputting $X$
4: for each $v \in V$ in the stream do   ▷ This entire loop requires only one pass
5:     if $v \in X$ then
6:         for each $(v, w) \in E$ in the stream do
7:             if $w \in V'$ then Output $(v, w)$ as part of the kernel
8:         $V' \leftarrow V' \cup \{v\}$
9:     else   ▷ $v \notin X$
10:        for each $(Y^+, Y^-, x, y) \in Z$ do   ▷ Reset local counters
11:            $y \leftarrow 0$
12:        Store an edge set $S \leftarrow \emptyset$   ▷ Reset local edge memory
13:        for each $(v, w) \in E$ in the stream do
14:            for each $(Y^+, Y^-, x, y) \in Z$ where $x \leq r$ and $y \geq 0$ do   ▷ Count ‘correct’ partitions
15:                if $w \in Y^+$ then $y \leftarrow y + 1$
16:                if $w \in Y^-$ then $y \leftarrow -1$
17:            $S \leftarrow S \cup \{(v, w)\}$.   ▷ If we mark $v$, then $(v, w)$ needs to be added
18:        if $\exists (Y^+, Y^-, x, y) \in Z$ where $y = |Y^+|$ then   ▷ Mark $v$ and output what we can
19:            $x \leftarrow x + 1$   ▷ Increment $x$ so that this partition marks at most $r$ vertices
20:            Output $S$ as part of the kernel
```

if we want the algorithm itself to store the entire kernel, this might increase the memory use, as we would have to store all edges contained in the kernel. The same goes for if we would want to output the kernel as an AL stream (which is how our input is provided). Let us shortly analyse how much memory would be needed for this. For a vertex cover X in a graph G, saving $G[X]$ entirely can take up to $\mathcal{O}(|X|^2 \log n)$ bits. Next to the vertex cover, the output kernel consists of $\mathcal{O}(r \cdot |X|^c)$ vertices, each of degree at most $|X|$, as these vertices are not part of the vertex cover. Therefore, the total memory use with this approach can be $\mathcal{O}((|X|^2 + r \cdot |X|^{c+1}) \log n)$ bits.

Applying ReduceStr to general problems

Next, we translate the general results provided in [42, Chapter 4] into the streaming setting using Theorem 4.2 (so using ReduceStr in place of Reduce).

The first general result is for Π-free Deletion problems. In the following, we call a graph G *vertex-minimal* with respect to Π if $G \in \Pi$ and for all $S \subsetneq V(G)$, $G[S] \notin \Pi$. We present the following theorem as an adaptation of [42, Theorem 4.1].

Theorem 4.3. *If* Π *is a graph property such that:*

(i) Π *is characterized by* c_Π *adjacencies,*
(ii) every graph in Π *contains at least one edge, and*
(iii) there is a non-decreasing polynomial $p : \mathbb{N} \to \mathbb{N}$ *such that all graphs* G *that are vertex-minimal with respect to* Π *satisfy* $|V(G)| \leq p(K)$,

then Π-free Deletion [VC] *admits a kernel on* $\mathcal{O}((K + p(K))K^{c_\Pi})$ *vertices in the* AL *streaming model using one pass and* $\mathcal{O}((K + K^{c_\Pi}) \log(n))$ *bits of space.*

Proof. Combine Theorem 4.2 with the proof of [42, Theorem 4.1], where instead of calling Reduce(G, X, $\ell + p(|X|)$, c_Π) we call ReduceStr(G, X, $\ell + p(|X|)$, c_Π). □

We note the the preconditions of Theorem 4.3 are both necessary and sufficient to achieve the stated result. Further motivation for their necessity can be found in [42, Chapter 4].

Let us go over a few examples that make use of Theorem 4.3. Consider Cluster Vertex Deletion parameterized by vertex cover. This problem is exactly Π-free Deletion [VC] where $\Pi = \{P_3\}$, as any P_3-free graph can only contain clusters. Notice that $c_\Pi = 2$ and $p(K) = 3$ suffice to meet the demands of Theorem 4.3. This implies that, given that we already have a vertex cover for our graph, we have a one-pass streaming algorithm for CVD in the AL model using $\mathcal{O}(K^2 \log n)$ space (or $\mathcal{O}(K^3 \log n)$ bits of space to save the kernel). Considering that this algorithm is a simple adaptation of known results, it is interesting to observe that it compares quite well to e.g. a more recent result of Bishnu *et al.* [8], who show that CVD admits a one-pass (randomized) streaming algorithm in the AL model using $\mathcal{O}(K^2 \log^4(n))$ space when parameterized by the size K of a vertex cover. Note that the kernel sizes differ, as the kernel given by Bishnu *et al.* [8] uses $\mathcal{O}(K^2 \log^2 n)$ bits of space, while the kernel from Algorithm 2 uses $\mathcal{O}(K^3 \log n)$ bits of space.

Next consider Triangle Deletion parameterized by vertex cover. This problem is exactly Π-free Deletion [VC] where $\Pi = \{C_3\}$. Once again, it is easy to observe that $c_\Pi = 2$, and $p(K) = 3$ suffice to meet the demands of Theorem 4.3. This means that, given that we already have a vertex cover, we have a one-pass streaming algorithm for TD in the AL model using $\mathcal{O}(K^3 \log n)$ bits of space (if we save the entire kernel). Once again, this result compares very well to a recent result of Bishnu *et al.* [8], who show that TD admits a one-pass streaming algorithm in the AL model using $\mathcal{O}(K^3)$ words of space when parameterized by the size K of a vertex cover.

The second general result is for finding largest induced subgraphs.

Largest Induced Π-Subgraph [VC]
Input: A graph G with a vertex cover X, and an integer $\ell \geq 1$.
Parameter: The size $K := |X|$ of the vertex cover.
Question: Is there a set $P \subseteq V(G)$ of size at least ℓ such that $G[P] \in \Pi$?

With this definition, we give the following theorem as an adaptation of [42, Theorem 4.2].

Theorem 4.4. *If* Π *is a graph property such that:*

(i) Π *is characterized by* c_Π *adjacencies, and*
(ii) there is a non-decreasing polynomial $p : \mathbb{N} \to \mathbb{N}$ *such that all graphs* $G \in \Pi$ *satisfy* $|V(G)| \leq p(K)$,

then Largest Induced Π-Subgraph [VC] *admits a kernel on* $\mathcal{O}(p(K) \cdot K^{c_\Pi})$ *vertices in the* AL *streaming model using one pass and* $\mathcal{O}((K + K^{c_\Pi}) \log(n))$ *bits of space.*

Proof. Combine Theorem 4.2 with the proof of [42, Theorem 4.2], where instead of calling Reduce(G, X, $p(|X|)$, c_Π) we call ReduceStr(G, X, $p(|X|)$, c_Π). □

Examples of problems fitting in the Largest Induced Π-Subgraph [VC] category, that are also characterized by few adjacencies, are Long Cycle, Long Path, and H-Packing.

The third general result is for graph partitioning problems.

Partition into q Disjoint Π-Free Subgraphs [VC]
Input: A graph G with a vertex cover X.
Parameter: The size $K := |X|$ of the vertex cover.
Question: Is there a partition of the vertex set into q sets $S_1 \cup S_2 \cup \ldots \cup S_q$ such that for each $i \in [q]$ the graph $G[S_i]$ does not contain a graph in Π as an induced subgraph?

Note that q is regarded a constant.

Theorem 4.5. *If* Π *is a graph property such that:*

(i) Π *is characterized by* c_Π *adjacencies, and*
(ii) there is a non-decreasing polynomial $p : \mathbb{N} \to \mathbb{N}$ *such that all graphs such that all graphs* G *that are vertex-minimal with respect to* Π *satisfy* $|V(G)| \leq p(K)$,

then Partition into q Disjoint Π-Free Subgraphs [VC] *admits a kernel on* $\mathcal{O}(p(K) \cdot K^{q \cdot c_\Pi})$ *vertices in the* AL *streaming model using one pass and* $\mathcal{O}((K + K^{q \cdot c_\Pi}) \log(n))$ *bits of space.*

Proof. Combine Theorem 4.2 with the proof of [42, Theorem 4.3] and [42, Lemma 4.2], where instead of calling Reduce(G, X, $q \cdot p(|X|)$, $q \cdot c_\Pi$) we call ReduceStr(G, X, $q \cdot p(|X|)$, $q \cdot c_\Pi$). □

Examples of problems fitting in the Partition into q Disjoint Π-Free Subgraphs [VC] category, that are also characterized by few adjacencies, are Partition into q Independent Sets, Partition into q Cliques, Partition into q Planar Graphs, and Partition into q Forests.

Theorem 4.3, Theorem 4.4, and Theorem 4.5 illustrate the broad range of problems that can be tackled using ReduceStr. However, not all Π can be characterized by few adjacencies. Next, we will see an improved characterization that leads to more results.

4.2.2 Kernel for Characterization by Low-Rank Adjacencies

More recently, Jansen and Kroon have given a kernel similar to the above Reduce (see Algorithm 1), with a different characterization [41]. This new result provides a kernelization algorithm for a more general set of problems, as it asks for a different characterization of the graph class Π. As the adaptation of Reduce to the streaming setting worked out well, we can wonder whether the same can be said for this new algorithm.

As before, we start by providing the definition of a new characterization, characterization by low-rank adjacencies. For this, however, we need the definition of a c-rank incidence vector.

Definition 4.6. ([41, Definition 5]) Let G be a graph with vertex cover X and let $c \in \mathbb{N}$. Let $Q', R' \subseteq X$ such that $|Q'| + |R'| \leq c$ and $Q' \cap R' = \emptyset$. We define the *c-incidence vector* $inc^{c,(Q',R')}_{(G,X)}(u)$ for a vertex $u \in V(G) \setminus X$ as a vector over $\mathbb{F}_2$ that has an entry for each $(Q, R) \subseteq X \times X$ with $Q \cap R = \emptyset$ such that $|Q| + |R| \leq c$, $Q' \subseteq Q$ and $R' \subseteq R$. It is defined as follows:

$$\text{inc}^{c,(Q',R')}_{(G,X)}(u)[Q,R] = \begin{cases} 1 & \text{if } N_G(u) \cap Q = \emptyset \text{ and } R \subseteq N_G(u), \\ 0 & \text{otherwise.} \end{cases}$$

The superscript (Q', R') is dropped when $Q' = R' = \emptyset$. Note that the order of the coordinates of the vector is fixed, but not explicit, as any order suffices. Therefore, we can also sum such incidence vectors coordinate-wise.

With this, we can give the defintion of a graph property being characterized by rank-c adjacencies.

Definition 4.7. ([41, Definition 7]) Let $c \in \mathbb{N}$ be a natural number. A graph property Π is characterized by rank-c adjacencies if the following holds. For each graph H, for each vertex cover X of H, for each set $D \subseteq V(H) \setminus X$, for each $v \in V(H) \setminus (D \cup X)$, if

- $H - D \in \Pi$, and
- $\text{inc}^c_{(H,X)}(v) = \sum_{u \in D} \text{inc}^c_{(H,X)}(u)$ when evaluated over $\mathbb{F}_2$,

then there exists $D' \subseteq D$ such that $H - v - (D \setminus D') \in \Pi$. If there always exists such set D' of size 1, then we say Π is characterized by rank-c adjacencies with singleton replacements.

Jansen and Kroon note that the intuition here is that if we have a set D such that $H - D \in \Pi$, and the c-incidence vectors of D sum to the vector of some vertex v over $\mathbb{F}_2$, then there exists $D' \subseteq D$ such that removing v from $H - D$ and adding back D' results in a graph that is still contained in Π. We can notice how there is some similarity in essential and non-essential adjacencies for occurrences of graphs in Π when comparing this characterization to that of few adjacencies (see Definition 4.1).

We will give the kernelization algorithm as given by Jansen and Kroon [41] next, but first, we recall a linear algebraic definition, that of a *basis.* As Jansen and Kroon state [41], "a *basis* of a set S of d-dimensional vectors over a field $\mathbb{F}$ is a minimum-size subset $B \subseteq S$ such that all $\vec{v} \in S$ can be expressed as linear combinations of elements of B, i.e., $\vec{v} = \sum_{\vec{u} \in B} \alpha_{\vec{u}} \cdot \vec{u}$ for a suitable choice of coefficients $\alpha_{\vec{u}} \in \mathbb{F}$. When working over the field $\mathbb{F}_2$, the only possible coefficients are 0 and 1, which gives a basis B of S the stronger property that any vector $\vec{v} \in S$ can be written as $\sum_{\vec{u} \in B'} \vec{u}$, where $B' \subseteq B$ consists of those vectors which get a coefficient of 1 in the linear combination".

The essence of the kernel comes down to computing the basis of a set of incidence vectors of the remaining graph and adding vertices corresponding to the basis to the kernel. We give this kernel here as Algorithm 3.

Algorithm 3 Low-Rank Reduce(Graph G, Vertex Cover $X \subseteq V(G)$, $\ell \in \mathbb{N}$, $c \in \mathbb{N}$) [41, Algorithm 1]

```
1: Let $Y_1 := V(G) \setminus X$
2: for $i \leftarrow 1$ to $\ell$ do
3:     Let $V_i = \{\text{inc}^c_{(G,X)}(y) \mid y \in Y_i\}$ and compute a basis $B_i$ of $V_i$ over $\mathbb{F}_2$.
4:     For each $\vec{v} \in B_i$, choose a unique vertex $y_{\vec{v}} \in Y_i$ such that $\vec{v} = \text{inc}^c_{(G,X)}(y_{\vec{v}})$.
5:     Let $A_i := \{y_{\vec{v}} \mid \vec{v} \in B_i\}$ and $Y_{i+1} = Y_i \setminus A_i$.
6: return $G[X \cup \bigcup_{i=1}^{\ell} A_i]$
```

Jansen and Kroon show that Algorithm 3 runs in polynomial time in ℓ and the size of G (for a constant c), and returns a graph on $\mathcal{O}(|X| + \ell \cdot |X|^c)$ vertices. We will now adapt Algorithm 3 to the streaming setting, and conclude a streaming equivalent of [41, Theorem 9].

Adapting Low-Rank Reduce

If we want to adapt Algorithm 3, Low-Rank Reduce, to the streaming setting, we are faced with a few challenges. For one, the set V_i consists of $\mathcal{O}(n)$ vectors, so saving this entire set is not desirable. We also have to consider how we can compute the basis of the set V_i if we do not want to save it. Luckily, the incidence vectors are computable from local information combined with the vertex cover, and computing a basis can be done incrementally by checking linear (in)dependence. With this small motivation, we give the adaptation of Low-Rank Reduce into the streaming setting, Low-Rank ReduceStr, in Algorithm 4.

Algorithm 4 LOW-RANK REDUCESTR(Graph G as an AL stream, Vertex Cover $X \subseteq V(G)$, $\ell \in \mathbb{N}$, $c \in \mathbb{N}$)

```
1: Let $A \leftarrow \emptyset$
2: for $i \leftarrow 1$ to $\ell$ do
3:     $A_i \leftarrow \emptyset$
4:     $B \leftarrow \emptyset$
5:     for each Vertex $v \in V \setminus (X \cup A)$ in the stream do        ▷ Entire loop in one pass
6:         Save the $\leq |X|$ adjacencies of $v$ in $X$ (until the next vertex)
7:         Let $\vec{v} \leftarrow \text{inc}^c_{(G,X)}(v)$        ▷ Can be computed from $X$ and the adjacencies of $v$
8:         If $\vec{v}$ is linearly independent w.r.t. $B$ over $\mathbb{F}_2$, do $B \leftarrow B \cup \{\vec{v}\}$ and $A_i \leftarrow A_i \cup \{v\}$.
9:     Let $A \leftarrow A \cup A_i$.
10: return $G[X \cup A]$
```

Theorem 4.8. *Algorithm 4 is a streaming equivalent of Algorithm 3, that is, given a graph G as an AL stream with a vertex cover X, and integer ℓ and a constant c, Algorithm 4 returns a graph on $\mathcal{O}(|X| + \ell \cdot |X|^c)$ vertices as an AL stream that could be the output of Algorithm 3 given the same input. Algorithm 4 uses $\ell + 1$ passes and $\mathcal{O}((|X| + \ell \cdot |X|^c)\log n + |X|^{2c})$ bits of memory.*

Proof. Let us first show the equivalence of Algorithm 4 to Algorithm 3. Let G be a graph with vertex cover X, and let ℓ and c be integers. Algorithm 4 and Algorithm 3 both do ℓ iterations over some process over all vertices, excluding the vertices in X and a set A (for both algorithms, $A = \bigcup_{i=1}^{\ell} A_i$). With these vertices, Algorithm 3 computes the incidence vector of each of them, and then computes a basis for this set. The new vertices added to A are then vertices with incidence vectors equivalent to those in the basis. Algorithm 4 computes the incidence vectors of these vertices one vertex at a time. It then checks whether the current incidence vector is linearly independent of a set B, and if so, the incidence vector is added to B. We can see that B must consist of a basis of all incidence vectors seen so far. Therefore, after Algorithm 4 has seen every vertex (excluding those in X and A), B is a basis that could also be found by Algorithm 3 for the same set of vertices. We conclude that in every iteration, Algorithm 4 adds to A a set of vertices which Algorithm 3 can also add to A in the corresponding iteration. As both algorithms return $G[X \cup A]$, the output of Algorithm 4 can also be an output of Algorithm 3.

As the incidence vector of a vertex v can be computed from the adjacencies of v together with X alone, and linear (in)depence checking only requires the vectors to be in memory, we only require one pass for each $1 \leq i \leq \ell$. Another pass is used to compute the output, as we can produce an AL stream corresponding to $G[X \cup A]$ by simply using a pass and output only those edges between vertices in $X \cup A$. Therefore, Algorithm 4 uses $\ell + 1$ passes over the stream.

In terms of memory, Algorithm 4 stores A, B, X, adjacencies of a vertex v, and an incidence vector of v, $\vec{v}$. Clearly, the memory used by the adjacencies of v and $\vec{v}$ is dominated by saving X and B in memory. The computation of $\vec{v}$ also requires no more memory than X uses, as it iterates subsets of X to compute entries of the vector. Saving X requires $\widetilde{\mathcal{O}}(|X|)$ bits of memory. B consists of at most $\mathcal{O}(|X|^c)$ vectors, and each vector consists of $\mathcal{O}(|X|^c)$ bits, as motivated by Jansen and Kroon [41, Proposition 8]. It follows that B requires $\mathcal{O}(|X|^{2c})$ bits of space. The set A consists of at most $\mathcal{O}(\ell \cdot |X|^c)$ vertices, as in every iteration B contains at most $\mathcal{O}(|X|^c)$ vectors. We conclude that the total memory usage is $\mathcal{O}((|X| + \ell \cdot |X|^c)\log n + |X|^{2c})$ bits. □

With Theorem 4.8 we are ready to give the streaming equivalent of [41, Theorem 9].

Theorem 4.9. *If Π is a graph property such that:*

(i) Π *is characterized by rank-c adjacencies,*
(ii) every graph in Π *contains at least one edge, and*
(iii) there is a non-decreasing polynomial $p : \mathbb{N} \to \mathbb{N}$ *such that all graphs* G *that are vertex-minimal with respect to* Π *satisfy* $|V(G)| \leq p(K)$*,*

then Π-free Deletion [VC] *in the AL streaming model admits a kernel on* $\mathcal{O}((K+p(K)) \cdot K^c)$ *vertices using* $K + p(K) + 2$ *passes and* $\mathcal{O}((K + p(K)) \cdot K^c \log n + K^{2c})$ *bits of memory.*

Proof. See the proof of [41, Theorem 9], where instead of Low-Rank Reduce$(G, X, \ell := k + 1 + p(|X|), c)$ we call Low-Rank ReduceStr$(G, X, \ell := k + 1 + p(|X|), c)$. By Theorem 4.8 the theorem follows. □

Let us shortly list some implications of Theorem 4.9, which consist of some problems admitting streaming kernels. These results are derived from the results by Jansen and Kroon [41, Section 4].

The first result is for Perfect Deletion [VC]. Perfect Deletion [VC] is Π-free Deletion [VC] where Π is the set of all graphs that contain an odd hole or an odd anti-hole. An *odd hole* is a cycle consisting of an odd number of vertices, and an *odd anti-hole* is the complement graph of an odd hole.

Theorem 4.10. Perfect Deletion [VC] *in the AL streaming model admits a kernel on* $\mathcal{O}(K^5)$ *vertices using* $\mathcal{O}(K)$ *passes and* $\mathcal{O}(K^5 \log n + K^8)$ *bits of memory.*

Proof. See [41, Theorem 19], but instead of applying [41, Theorem 9] we apply Theorem 4.9. □

The second result is for AT-free Deletion [VC]. This is Π-free Deletion [VC] where Π is the set of all graphs that contain an asteroidal triple. An *asteroidal tiple* is a set of three vertices where every two vertices in the triple are connected by a path that avoids the neighbourhood of the third.

Theorem 4.11. AT-free Deletion [VC] *in the AL streaming model admits a kernel on* $\mathcal{O}(K^9)$ *vertices using* $\mathcal{O}(K)$ *passes and* $\mathcal{O}(K^9 \log n + K^{16})$ *bits of memory.*

Proof. See [41, Theorem 21], but instead of applying [41, Theorem 9] we apply Theorem 4.9. □

The third result is for Interval Deletion [VC]. This is Π-free Deletion [VC] where Π is the set of all graphs that contain either an asteroidal triple or an induced cycle of length at least 4, or both.

Theorem 4.12. Interval Deletion [VC] *in the AL streaming model admits a kernel on* $\mathcal{O}(K^9)$ *vertices using* $\mathcal{O}(K)$ *passes and* $\mathcal{O}(K^9 \log n + K^{16})$ *bits of memory.*

Proof. See [41, Theorem 22], but instead of applying [41, Theorem 9] we apply Theorem 4.9. □

The fourth result is for Wheel-free Deletion [VC]. This is Π-free Deletion [VC] where Π is the set of all graphs that contain a wheel of size at least 3. A *wheel* of size $n \geq 3$ is a set of $n + 1$ vertices, where n vertices form a cycle, and one vertex is connected to all vertices on the cycle (the center of the wheel).

Theorem 4.13. Wheel-free Deletion [VC] *in the AL streaming model admits a kernel on* $\mathcal{O}(K^5)$ *vertices using* $\mathcal{O}(K)$ *passes and* $\mathcal{O}(K^5 \log n + K^8)$ *bits of memory.*

Proof. See [41, Theorem 24], but instead of applying [41, Theorem 9] we apply Theorem 4.9. □

It is interesting to note that the above problems are not characterized by few adjacencies, but only characterized by rank-c adjacencies, therefore requiring Algorithm 4 to admit a streaming kernel.

4.3 A Direct FPT Approach

In the previous section we have seen how very general problems admit streaming kernels in the Adjacency List model. It is known in folklore that a kernel implies an FPT algorithm, and so we know that these problems admit streaming FPT algorithms. However, we can wonder whether pursuing a direct FPT algorithm rather than through a kernel can deliver different results than those implied by the streaming kernels. This is motivated by the fact that Chitnis and Cormode [17] found a direct FPT algorithm for VERTEX COVER using $\mathcal{O}(2^k)$ passes and only $\widetilde{\mathcal{O}}(k)$ space in contrast to the kernel of Chitnis *et al.* [18] using one pass and $\widetilde{\mathcal{O}}(k^2)$ space. Therefore, this section aims to explore the pass/memory trade-off for Π-FREE DELETION [VC], by attempting to find a direct algorithm for the same cases as Theorem 4.3.

4.3.1 P_3-free Deletion

To start exploring this approach for Π-FREE DELETION [VC], let us look at a specific case. We start with the scenario where $\Pi = \{P_3\}$, which means we consider the problem CLUSTER VERTEX DELETION parameterized by vertex cover.

The general idea of the algorithm is to branch on what part of the given vertex cover should be in the solution. This gives us a lot of information to work with, as within a branch we do not allow any vertex of the other part of the vertex cover to be part of the solution. What remains is some case analyses where either one or two vertices of a P_3 lie outside the vertex cover, for which we deterministically know which vertices have to be removed to make the graph P_3-free. Below, the idea of this algorithm is immediately adapted to the streaming setting, and so we present a streaming algorithm for CLUSTER VERTEX DELETION [VC] in the AL streaming model.

To handle branching formally, we need the same dictionary ordering structure as used by Chitnis and Cormode [17], which we define here.

Definition 4.14. ([17, Definition 9]) Let $U = \{u_1, u_2, \ldots, u_n\}$ and $k \leq n$. Let $\mathcal{U}_{\leq k}$ denote the set of all $\sum_{i=0}^{k} \binom{|U|}{i}$ subsets of U which have at most k elements, and let $\textsc{Dict}_{\mathcal{U}_{\leq k}}$ be the dictionary ordering on $\mathcal{U}_{\leq k}$. Given a subset $X \in \mathcal{U}_{\leq k}$, let $\textsc{Dict}_{\mathcal{U}_{\leq k}}(\textsc{Next}(X))$ denote the subset that comes immediately after X in the ordering $\textsc{Dict}_{\mathcal{U}_{\leq k}}$. We denote the last subset in the dictionary order of $\mathcal{U}_{\leq k}$ by $\textsc{Last}(\mathcal{U}_{\leq k})$, and similarly the first subset as $\textsc{First}(\mathcal{U}_{\leq k})$, and use the notation that $\textsc{Dict}_{\mathcal{U}_{\leq k}}(\textsc{Next}(\textsc{Last}(\mathcal{U}_{\leq k}))) = \spadesuit$. Similarly, we define $\mathcal{U}_k$ as the set of all $\binom{|U|}{k}$ subsets of U with exactly k elements, and analogously define the dictionary ordering on this set.

For a look into the inner workings of dictionary orderings and an analysis of their memory use, see Appendix A.

The algorithm for CLUSTER VERTEX DELETION [VC] in the AL streaming model is now given as Algorithm 5. Remember that our aim is to use $o(K^2)$ space, as applying Theorem 4.3 to CLUSTER VERTEX DELETION [VC] gives us an $\widetilde{\mathcal{O}}(K^2)$ space kernel. In this regard, we note that Algorithm 5 can be improved to use less passes over the stream by saving the entire vertex cover with edges to memory, but doing so would increase its space usage to $\widetilde{\mathcal{O}}(K^2)$.

Algorithm 5 P_3-free Deletion(Graph $G = (V, E)$ given as a stream in the AL model, integer ℓ, Vertex Cover $X \subseteq V(G)$)

```
1: $S \leftarrow$ First($\mathcal{X}_{\leq \ell}$)
2: while $S \in \mathcal{X}_{\leq \ell}, S \neq ♠$ do
3:     $Y \leftarrow X \setminus S$                    ▷ $Y$ is the part of the vertex cover not in the solution $S$
4:     $S' \leftarrow S$                               ▷ If $S'$ ever exceeds size $\ell$, move to the next $S$
5:     $P \leftarrow$ First($\mathcal{Y}_2$)
6:     while $P = (y_1, y_2) \in \mathcal{Y}_2, P \neq ♠$ do          ▷ We enumerate all pairs in $Y$
7:         for each Vertex $v \in Y \setminus P$ do
8:             If $v$ and $P$ form a $P_3$, $Y$ is invalid, move to the next $S$      ▷ Requires a pass
9:         $P \leftarrow$ Dict$_{\mathcal{Y}_2}$(Next($P$))
10:    $P \leftarrow$ First($\mathcal{Y}_2$)
11:    while $P = (y_1, y_2) \in \mathcal{Y}_2, P \neq ♠$ do          ▷ We enumerate all pairs in $Y$
12:        if $y_1 y_2$ is an edge then
13:            for each Vertex $v \in V \setminus (X \cup S')$ in the stream do     ▷ Entire loop in one pass
14:                if Either $vy_1$ or $vy_2$ is present and the other is not then $S' \leftarrow S' \cup \{v\}$
15:        else
16:            for each Vertex $v \in V \setminus (X \cup S')$ in the stream do     ▷ Entire loop in one pass
17:                if Both $vy_1$ and $vy_2$ are present then $S' \leftarrow S' \cup \{v\}$
18:        $P \leftarrow$ Dict$_{\mathcal{Y}_2}$(Next($P$))
19:    for each $y \in Y$ do
20:        $b \leftarrow False$
21:        for each Vertex $v \in V \setminus (X \cup S')$ in the stream do         ▷ Entire loop in one pass
22:            if The edge $vy$ is present and $b = False$ then $b \leftarrow True$
23:            else if The edge $vy$ is present then $S' \leftarrow S' \cup \{v\}$
24:    if $|S'| \leq \ell$ then return $S'$
25:    $S \leftarrow$ Dict$_{\mathcal{X}_{\leq \ell}}$(Next($S$))
26: return NO                                         ▷ No branch resulted in a solution
```

Theorem 4.15. *Algorithm 5 gives a solution to* Cluster Vertex Deletion [VC] *if it exists, and returns NO otherwise, using $\mathcal{O}(2^K K^2)$ passes and $O(K \log n)$ bits space, where $|X| = K$, the size of the vertex cover.*

Proof. Let us first reason that the number of passes and memory use are as stated. Let X be the provided vertex cover, and let $|X| = K$. The number of different sets S can take is bounded by 2^K. The first and second loop enumerate all pairs of vertices in the vertex cover, and use a single pass per pair to detect a P_3, which gives us at most K^2 passes. The last loop enumerates all K vertices in the vertex cover and uses one pass per iteration. Therefore, the total number of passes is bounded by $\mathcal{O}(2^K K^2)$.

In terms of memory, we save the sets S, S', Y, X, P and some separate bits for boolean logic. Notice that S and S' never exceed $\mathcal{O}(\ell \log n)$ bits of memory, as we stop whenever it does. Both X and Y are bounded by the size of the vertex cover, which is $\mathcal{O}(K \log n)$ bits. P is negligible in comparison to X and Y, as are the separate bits. Therefore, assuming $\ell \leq K$ (which we assume because else the vertex cover itself is a trivial solution), the memory use is $\mathcal{O}(K \log n)$ bits.

Let us now show the correctness of the algorithm. The main idea of the algorithm is to branch on what part of the vertex cover is contained in the solution S'. This is modelled through the

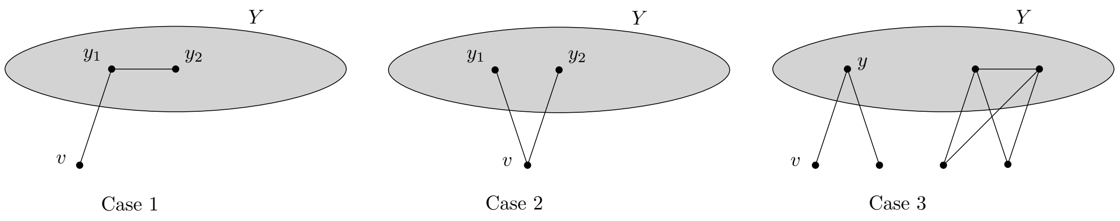

Figure 2: The different cases how a P_3 can exists with respect to Y, part of the vertex cover. Notice that the case where the entire P_3 is contained in Y is not included here. Case 3 assumes there are no Case 1 or Case 2 P_3's in the graph anymore.

use of the sets Y and S, where in each branch, we cannot add vertices in Y to S'. Therefore, we first check whether Y fully contains a P_3, and if one is found we stop, as we may not delete any vertex of this P_3 in this branch. For a fixed pair (v, w) in the vertex cover, checking for a P_3 that contains v and w only takes one pass because the only necessary information is the adjacencies of v and w towards another vertex, which is provided in the stream local to that vertex (see also Case 1 and Case 2 in the following analysis).

What remains is a careful analysis of the different cases of the structure of P_3's with respect to Y. An illustration is given in Figure 2. The loop of line 11 considers all pairs of vertices in Y. There are two cases we are interested in: Case 1 and Case 2 in Figure 2. If we look at a single pair of vertices y_1 and y_2 either there is an edge between them (Case 1) or a non-edge (Case 2). These two vertices can then form a P_3 with any vertex outside Y in a very specific manner, which the algorithm looks for. It is then trivial that the one vertex outside Y has to be removed to make the graph P_3 free if a P_3 is found.

If there are no Case 1 or Case 2 P_3's in the graph any more, we move on to Case 3. Note that this is the only remaining way a P_3 can be in the graph at all, as vertices outside Y and S cannot have edges between them, because $Y \subseteq X$ is (part of) a vertex cover. In Case 3 we might at first be worried that we do not know which of the two vertices outside Y to remove, as one might lead to a solution and the other not. Let y_1, v, w form a Case 3 P_3, where $y_1 \in Y$. Let us consider the scenario where v has another adjacency $y_2 \in Y$. Because there are no Case 2 P_3's, y_1 and y_2 must be adjacent. Because there are no Case 1 P_3's, w must now also be adjacent to y_2. This means the structure extends as illustrated on the right in Case 3 in Figure 2. We can observe that we need to delete all but one of the vertices attached to y, which is what the algorithm does. It does not matter which vertex we do not delete, as this vertex forms triangles if it has multiple adjacencies. Therefore, after these cases have all been handled, no induced P_3's remain in the graph. If during the process S' never exceeded size ℓ, this means we have found a solution; otherwise, we move on to the next branch.

By the above reasoning, if there exists a solution of size at most ℓ for the Cluster Vertex Deletion [VC] problem, then this solution contains some subset of the vertex cover X, which corresponds to some branch in the algorithm. As the removal of vertices is deterministic in each branch, and there exists a solution, the algorithm must find a solution too in that branch. If there exists no solution of size at most ℓ, then there exists no subset of vertices S' such that $G \setminus S'$ is induced P_3 free, and so in each branch of the algorithm S' will exceed size ℓ at some point, which results in the algorithm returning NO. □

Now that we have seen Algorithm 5, it is good to closely look at some of its features, because they can provide relevant insights. One of the first noticeable features is that this algorithm works on the Adjacency List model. This is arguably the 'weakest' model, as it forces the stream to have a very specific structure, but its use is necessary here. The only reason we can do the loops over all vertices in one pass, is because we know we get all adjacencies of a vertex towards the vertex cover together. This allows us to immediately (locally) make some decision without having to save information to memory. If we were to use the EA or VA model, where there is no guarantee that this will happen, we have two options: Either save for every vertex some information, or use for every vertex multiple passes to get the same information. Both of these options clearly worsen the performance of the algorithm drastically. This difference indicates a quite significant distinction between these models and the amount of information they provide.

Another noticeable feature is the heavy use of dictionary orderings, as defined in Definition 4.14. Their use is easily motivated by their purpose, enumerating certain sets while using little memory to do so. This is clearly suited to use in the streaming model, where with branching purposes we often want to enumerate sets while using little memory. The use of dictionary orderings in the streaming setting was introduced by Chitnis and Cormode [17], who illustrate its potency with two algorithms for Vertex Cover heavily relying on these dictionary orderings as well.

Lastly, let us note the implications this algorithm has outside of the streaming setting. We can easily implement Algorithm 5 without making use of a stream, and simply executing the foreach-statements as such. It is not difficult to see that this would lead to an algorithm for Cluster Vertex Deletion [VC] running in $\mathcal{O}(2^K \cdot K^2 \cdot (n+m))$ time. Note that this is the same as the number of passes Algorithm 5 does in the streaming setting, except for a difference in a factor $n+m$. This is not strange, seeing as a single pass of the stream corresponds to doing something for every vertex or edge. In Algorithm 5 we can observe that every edge adjacent to a vertex is considered a constant number of times for that vertex, so the complexity follows. So we have also found an FPT algorithm for Cluster Vertex Deletion [VC].

4.3.2 H-free Deletion

Let us generalize the above ideas to a more general form, H-free Deletion parameterized by vertex cover. This is Π-free Deletion [VC] where $\Pi = \{H\}$, a single graph. Seeing the success of the P_3 case might incline us to think that carefully analysing the H-free case can also result in an effective algorithm. However, in the general case it is not possible to use the structure of H so elaborately as in the P_3 case. That is, the case analysis done in Algorithm 5 works so effectively because the structure of a P_3 is very simple and very local. What this leads to is that the running time of the FPT algorithm for H-free Deletion [VC], and so the number of passes for the streaming algorithm (these are closely related), is quite significantly worse in comparison to the P_3 case. Nonetheless, these results are mentionable, as they are far more general than the P_3 case.

The main issues arising with the more general H-free Deletion [VC] come from two factors: It requires some effort to find an occurrence of H in the graph, and if we find an H, it is certainly not deterministic which deletion will lead to a solution. The first problem causes just generally a bigger running time (possibly hidden in constants) and so more passes in the streaming setting. The second issue is more difficult, as it means we will have to do more branching to find a solution. And, as one can imagine, branching on possible deletions within branches of the vertex cover leads to larger running times.

We first provide an FPT algorithm for H-free Deletion [VC] (not in the streaming setting), see Algorithm 6.

Let us first analyse the subroutine FindH of Algorithm 6.

Lemma 4.16. *In Algorithm 6,* FindH *is a function that, given a graph $G = (V, E)$ with vertex cover X, graph H with at least one edge, and sets S, $Y \subseteq X$, and integer i, finds an occurrence*

Algorithm 6 *H*-free Deletion FPT(Graph $G = (V, E)$, integer ℓ, Vertex Cover $X \subseteq V(G)$)

```
1: for each Partition of $X$ into $S, Y$ where $|S| \leq \ell$ do
2:     if $H$ is not contained in $Y$ then                ▷ Check all $\mathcal{O}(\binom{|X|}{|H|}|H|!)$ options
3:         if Branch($S$, $Y$, 1) then
4:             return YES                                  ▷ If any returns YES, we also return YES
5: return NO

6: function Branch(solution set $S$, forbidden set $Y \subseteq X$, integer $i$)
7:     $B \leftarrow$ FindH($S$, $Y$, $i$)                  ▷ Try to find an $H$ with $i$ vertices outside $Y$
8:     if $B = \emptyset$ and $i = |H|$ then return YES
9:     else if $B = \emptyset$ then Branch($S$, $Y$, $i + 1$)                      ▷ No $H$ found
10:    else if $|S| = \ell$ then return NO                 ▷ Found an $H$ but cannot remove it
11:    else
12:        for each $v \in B$ do
13:            if Branch($S \cup \{v\}$, $Y$, $i$) then return YES

14: function FindH(solution set $S$, forbidden set $Y \subseteq X$, integer $i$)
15:    for each Set $O$ of $i$ vertices of $H$ that can be outside $Y$ do      ▷ Check the needed $i^2$
   non-edges in $H$
16:        Denote $H' = H \setminus O$
17:        for each Occurrence of $H'$ in $Y$ do           ▷ Check all $\mathcal{O}(\binom{|X|}{|H|-i}(|H| - i)!)$ options
18:            $S' \leftarrow \emptyset$, $O' \leftarrow O$
19:            for each Vertex $v \in V \setminus (S \cup X)$ do
20:                Check the edges/non-edges towards $H' \in Y$
21:                if $v$ is equivalent to some $w \in O'$ for $H'$ then
22:                    $S' \leftarrow S' \cup \{v\}$, $O' \leftarrow O' \setminus \{w\}$
23:                    if $O' = \emptyset$ then return $S'$    ▷ We found an occurrence of $H$
24:    return $\emptyset$                                   ▷ No occurrence of $H$ found
```

of H in G such that this occurrence contains no vertices in S and $X \setminus Y$, and with $|V(H)| - i$ vertices contained in Y. FindH *runs in* $\mathcal{O}\left(\binom{h}{i}[i^2 + \binom{K}{h-i}(h-i)!((h-i)^2 + Kn + (h-i)in)]\right)$ *time, where $|V(H)| = h$, and $|X| = K$.*

Proof. Let us first show the correctness of FindH, that is, we show that FindH correctly finds occurrences of H where i vertices are outside Y with no vertices from S. The function starts by enumerating all sets of i vertices in H that can be outside Y. This requires all these i vertices to have no edges between them, which the algorithm checks. If the check is successful, it then enumerates all possibilities for the part of H inside Y. For each of these options, it enumerates all vertices outside X and S to find the vertices equivalent to those in O (we do not consider the vertices in S as these are already removed from the graph). Clearly, if it finds these vertices then it has found an occurrence of H with i vertices outside of Y. If it finds no occurrence and returns the empty set, let it be clear by the above motivation that the algorithm has considered all possibilities for such an H to occur, and therefore, there is none.

Let us analyse the running time of FindH. Checking all possible sets O takes $\mathcal{O}(\binom{h}{i}i^2)$ time resulting in at most $\mathcal{O}(\binom{h}{i})$ options for O. There are at most $\mathcal{O}(\binom{K}{h-i}(h-i)!)$ options for H' in Y: checking all of them costs $\mathcal{O}(\binom{K}{(h-i)}(h-i)!(h-i)^2)$ time. Then we take $\mathcal{O}((K + (h-i)i)n)$ time to, for each vertex, save adjacencies to H', and check whether it matches on of those in O. The $(h-i)$ factor is for checking adjacencies towards H'. Therefore, the running time of FindH is $\mathcal{O}\left(\binom{h}{i}[i^2 + \binom{K}{h-i}(h-i)!((h-i)^2 + Kn + (h-i)in)]\right)$. □

With Lemma 4.16 we can analyse Algorithm 6 in its entirety.

Theorem 4.17. *Algorithm 6 is an FPT algorithm for H-free Deletion [VC] using $\mathcal{O}(2^K h^K K^{h+1} h! h^2 n)$ time or alternatively $\mathcal{O}(2^K h^K K! K h! h^2 n)$ time, where $K = |X|$, the size of the vertex cover, and $|V(H)| = h$ and H contains at least one edge.*

Proof. Let us first go into detail on the correctness of the algorithm. Assume the algorithm returns YES for some instance G, H, ℓ, X where $|X| = K$ and $|V(H)| = h$. The only way the algorithm returns YES, is if in some partition of X into S and Y the Branch function returns YES. The Branch function only returns YES if any recursive call returns YES, or when $B = \emptyset$ and $i = h$. As the latter is the only base case, this must have occurred for this instance. As i starts at 1 and is only ever incremented, we can conclude that for every i at some point $B = \emptyset$ while $|S| \leq \ell$. The algorithm calls on FindH for every i to find if there is an occurrence of H with i vertices outside of $Y \cup S$ and $|V(H)| - i$ vertices in Y. By Lemma 4.16, FindH correctly finds occurrences of H where i vertices are outside Y. As the algorithm returned YES, FindH must have returned an empty set for each i at some point, and so no occurrences of H are present in the graph $G[V \setminus S]$ (otherwise, such an occurrence must have i vertices outside $Y \cup S$ for some i). This means that the algorithm is correct in returning YES.

For the other direction, assume that for an instance G, H, ℓ, X where $|X| = K$ there exists a smallest set S_{opt} such that $G[V \setminus S_{opt}]$ is H-free and $|S_{opt}| \leq \ell$. Then S_{opt} must contain some part of the vertex cover X, and as we enumerate all possibilities, the algorithm considers this option. As $G[V \setminus S_{opt}]$ is H-free, clearly, for every set B the function FindH finds, at least one vertex in B is also in S_{opt}. As we branch on each possibility of the vertices in B, the algorithm also considers exactly the option where the set S in the algorithm is a subset of S_{opt}. This means there is a branch where the algorithm terminates with $S = S_{opt}$, which means it returns YES as $G[V \setminus S_{opt}]$ is H-free. We conclude that the algorithm solves H-free Deletion correctly.

Let us analyse the running time of the algorithm. There are $\mathcal{O}(2^K)$ possible partitions of X into S and Y. Checking whether H is contained in Y takes $\mathcal{O}(\binom{K}{h}h!h^2)$ time. Because H

contains at least one edge, we can assume that $\ell \leq K$, as otherwise X is a trivial solution. The function BRANCH is called in worst case $\mathcal{O}(h^\ell) = \mathcal{O}(h^K)$ times (branching on at most h vertices each time). FINDH is called at most once for every i in every branch. By Lemma 4.16, FINDH runs in $\mathcal{O}\left(\binom{h}{i}[i^2 + \binom{K}{h-i}(h-i)!((h-i)^2 + Kn + (h-i)in)]\right)$ time. Here is where there is some variance in how we round these complexities, namely when concerning e.g. $\binom{K}{h}$. This is because we can both say $\binom{K}{h} = \mathcal{O}(K^h)$, and $\binom{K}{h} = \mathcal{O}(K!)$. Which of these is a tighter bound comes down to the value of h in comparison to K. The total complexity of the algorithm comes down to

$$\mathcal{O}\left(2^K\left(\binom{K}{h}h!h^2 + h^K\sum_{i=1}^{h}\binom{h}{i}[i^2 + \binom{K}{h-i}(h-i)!((h-i)^2 + Kn + (h-i)in)]\right)\right)$$

time, which we can shorten to either $\mathcal{O}(2^K h^K K^{h+1} h! h^2 n)$ or $\mathcal{O}(2^K h^K K! K h! h^2 n)$ time. □

Before we go into the translation of Algorithm 6 to the streaming model, let us discuss one of its shortcomings. A large part of the complexity of the algorithm comes from the twofold branching (on the vertex cover and on the possible deletions), but next to this, the function FINDH largely contributes to the complexity. We can ask ourselves whether or not this function can be made more efficient. This means we are interested in more efficient induced subgraph finding, which is also called induced subgraph isomorphism. This problem has been studied in the literature, with varying degrees of success. There are a couple of main issues with regard to applying such results to this algorithm. For one, many results focus on specific graph structures, e.g. finding r-regular induced subgraphs [48]. These results do not help us as we are interested in general structures. Another problematic factor is the common approach of using matrix multiplication. The issue with matrix multiplication is that it does not translate well to the streaming model, as often matrices require at least super-linear memory. An example of such an algorithm can be found in [43].

Let us now translate Algorithm 6 into the streaming model, see Algorithm 7. It should be clear that the functionality of Algorithm 7 is the same as that of Algorithm 6, but translated to the streaming model using as little memory as possible. Once again we make use of dictionary orderings, see Definition 4.14 for the formal definition.

Theorem 4.18. *Algorithm 7 is a streaming algorithm for H-FREE DELETION [VC] in the AL model using $\mathcal{O}(2^K h^{K+2} K^h h!)$ or alternatively $\mathcal{O}(2^K h^{K+2} K! h!)$ passes and $\mathcal{O}((K+h^2)\log n)$ bits of space, where $K = |X|$, the size of the vertex cover, and $|V(H)| = h$ and H contains at least one edge.*

Proof. Let it be clear from the algorithm that the approach to solving H-FREE DELETION has not changed from Algorithm 6. Therefore, if the graph stream is handled correctly, we can conclude that H-FREE DELETION is solved correctly. As we have H in memory, we only require to use passes of the stream to determine (parts of) X and G. The dictionary orderings require no passes because we have the vertices of X in memory. The only places where we require passes of the stream thus are when concerning edges of the vertex cover, and edges/vertices in the rest of the graph ($V \setminus X$). Notice that at such points in Algorithm 7 we correctly mention the use of a pass. The loop over all vertices in $V \setminus (S \cup Y)$ only requires one pass because of the use of the AL model. Therefore, the algorithm is a correct adaptation of Algorithm 6 to the streaming model.

What remains is to analyse the number of passes and memory use. Let us analyse the memory use per function. The entire algorithm keeps track of the vertex cover X, and the forbidden graph

Algorithm 7 H-FREE DELETION STREAM(Graph $G = (V, E)$ in the AL model, integer ℓ, Vertex Cover $X \subseteq V(G)$)

```
 1: $S \leftarrow$ FIRST($\mathcal{X}_{\leq \ell}$)
 2: while $S \in \mathcal{X}_{\leq \ell}, S \neq ♠$ do
 3:     $Y \leftarrow X \setminus S$
 4:     $S' \leftarrow S$
 5:     if $\neg$ CHECK($H$,$Y$) then
 6:         if BRANCH($S'$, $Y$, 1) then
 7:             return YES                        ▷ If any returns YES, we also return YES
 8:     $S \leftarrow$ DICT$_{\mathcal{X}_{\leq \ell}}$(NEXT($S$))
 9: return NO

10: function CHECK(set to find $H$, search space $Y$)        ▷ Tries to find an $H$ contained in $Y$
11:     $P \leftarrow$ FIRST($\mathcal{Y}_{|H|}$)
12:     while $P \in \mathcal{Y}_{|H|}, P \neq ♠$ do
13:         for each Permutation $p$ of the vertices of $H$ do
14:             Use a pass to check if $p$ matches $P$ ▷ Go to the next $p$ if some edges do not match
15:             if $p$ matches $P$ then return YES
16:         $P \leftarrow$ DICT$_{\mathcal{Y}_{|H|}}$(NEXT($P$))
17:     return NO

18: function BRANCH(solution set $S$, forbidden set $Y \subseteq X$, integer $i$)
19:     $B \leftarrow$ FINDH($S$, $Y$, $i$)                ▷ Try to find an $H$ with $i$ vertices outside $Y$
20:     if $B = \emptyset$ and $i = |H|$ then return YES
21:     else if $B = \emptyset$ then BRANCH($S$, $Y$, $i + 1$)                     ▷ No $H$ found
22:     else if $|S| = \ell$ then return NO                  ▷ Found an $H$ but cannot remove it
23:     else
24:         for each $v \in B$ do
25:             if BRANCH($S \cup \{v\}$, $Y$, $i$) then return YES

26: function FINDH(solution set $S$, forbidden set $Y \subseteq X$, integer $i$)
27:     for each Set $O$ of $i$ vertices of $H$ that can be outside $Y$ do      ▷ Check the needed $i^2$
    non-edges in $H$
28:         Denote $H' = H \setminus O$
29:         $P \leftarrow$ FIRST($\mathcal{Y}_{|H'|}$)
30:         while $P \in \mathcal{Y}_{|H'|}, P \neq ♠$ do
31:             for each Permutation $p$ of the vertices of $H'$ do
32:                 Use a pass to check if $p$ matches $P$
33:                 if $p$ matches $P$ then
34:                     $S' \leftarrow \emptyset$, $O' \leftarrow O$
35:                     for each Vertex $v \in V \setminus (S \cup Y)$ do        ▷ Entire loop in one pass
36:                         Check the edges/non-edges towards $H' \in Y$         ▷ Use $\mathcal{O}(|H'|) bits$
37:                         if $v$ is equivalent to some $w \in O'$ for $H'$ then
38:                             $S' \leftarrow S' \cup \{v\}$, $O' \leftarrow O' \setminus \{w\}$
39:                             if $O' = \emptyset$ then return $S'$          ▷ We found an $H$ occurrence
40:             $P \leftarrow$ DICT$_{\mathcal{Y}_{|H'|}}$(NEXT($P$))
41:     return $\emptyset$                                    ▷ No occurrence of $H$ found
```

H. Denote $|X| = K$ and $|V(H)| = h$. The vertex cover uses $\mathcal{O}(K)$ words, and because we save the entirety of H we use $\mathcal{O}(h^2)$ words. The main function uses the set S of at most ℓ elements of X, likewise S', and the set Y of size $\mathcal{O}(K)$. The function Check uses a set P of h elements, a permutation p of h elements, and $\mathcal{O}(1)$ bits to check if p matches P, as it can stop when it does not match. The function Branch uses a set B of at most h elements and increases the size of the set S, but only when it does not exceed ℓ elements. The function FindH uses sets O, H', P, O', p, all of at most h vertices. It also uses $\mathcal{O}(1)$ bits to check if p matches P, and $\mathcal{O}(h)$ bits to save the adjacencies to H' to check if some vertex matches one in O'. S' can only contain an element for each element in O'. Therefore, S' contains at most h elements. We can conclude the memory use of a single branch is bounded by $\mathcal{O}((K + h^2)\log n)$ bits. However, in this algorithm we branch on h options, which is not a constant. Therefore, to be able to return out of recursion when branching and continue where we left off, we need to save the set B, or recompute it when we return. Saving the sets B takes $\widetilde{\mathcal{O}}(hK)$ bits because we have at most K active instances. Alternatively, recomputing B adds a factor h to the number of passes. Seeing as we aim to be memory efficient, we opt for this second option here.

The number of passes used by the algorithm is closely aligned with the running time of Algorithm 6. There are only three places in which we use a pass of the stream, namely, line 14 in the function Check, and line 32 and line 35 in the function FindH. The loop of line 35 requires only one pass because the stream is given in the Adjacency List model. The number of passes is clearly dominated by the number of times the passes in line 32/35 are used. Now we can use the same analysis as for Algorithm 6, but make some nuances in the running time, as we have to distinguish running time which leads to more passes and running time which will be 'hidden' in the allowed unbounded computation. Consider the running time of FindH, as given by Lemma 4.16, $\mathcal{O}\left(\binom{h}{i}[i^2 + \binom{K}{h-i}(h-i)!((h-i)^2 + (h-i)in)]\right)$. The running time factors i^2 and $(h-i)^2$ are checks over some amount of edges, which will be hidden by the unbounded computation. The factor $(h-i)in$ comes from the finding of vertices that fit the current form of H, and can be done in one pass, which means this factor falls away as well. We now have that the FindH function costs us $\mathcal{O}\left(\binom{h}{i}\binom{K}{h-i}(h-i)!\right)$ passes. This can be shortened to $\mathcal{O}(K^h h!)$ by expanding the $\binom{h}{i}$ factor, and bounding $\binom{K}{h-i} = \mathcal{O}(K^h)$. We can also bound this by $\mathcal{O}(K!h!)$ as discussed before. As FindH is called for every $1 \leq i \leq h$ in every branch, we get an extra h factor in the total number of passes. Another factor h is added for recomputing B when returning out of recursion at every step. This means the total number of passes comes down to either $\mathcal{O}(2^K h^{K+2} K^h h!)$ or $\mathcal{O}(2^K h^{K+2} K! h!)$. □

4.3.3 Towards Π-free Deletion

The running time of Algorithm 6 and the number of passes of Algorithm 7 are not ideal. If we were to extend these algorithms to the general Π-free Deletion without further improvements the performance would be far from desirable. One of the issues is the dependence of the size of the graph $H \in \Pi$ we look for, h. Without any further analysis, we have no bound on h. However, we can look to the preconditions used by Jansen on Π in e.g. Theorem 4.3 for improvement. As a reminder, we are trying to achieve similar results to Theorem 4.3 through another approach. Therefore, we can freely use the same preconditions.

One of these preconditions is particularly interesting, namely the assumption that all graphs $H \in \Pi$ that are vertex-minimal with respect to Π have a size bounded by a polynomial in K, the size of the vertex cover. More specifically, for these graphs H we have that $|V(H)| \leq p(K)$, where $p(K)$ is some polynomial in the vertex cover size K. We can easily prove that it suffices to only remove vertex-minimal elements of Π to solve Π-free Deletion.

Lemma 4.19. *Let Π be some graph property, and denote the set of vertex-minimal graphs in Π with Π'. Let G be some graph and $S \subseteq V(G)$ some vertex set. Then $G[V(G) \setminus S]$ is Π-free if and only if $G[V(G) \setminus S]$ is Π'-free.*

Proof. Assume the preconditions in the lemma, and assume that $G[V(G) \setminus S]$ is not Π'-free. As $\Pi' \subseteq \Pi$, clearly, $G[V(G) \setminus S]$ is not Π-free.

Now assume that $G[V(G) \setminus S]$ is Π'-free. Assume there is some $H \in \Pi$ such that $H \in G[V(G) \setminus S]$ (that is, H is isomorphic to an induced subgraph of $G[V(G) \setminus S]$). As the graph is Π'-free, H is a non-vertex-minimal graph with respect to Π. By definition of vertex-minimal, removal of a specific set of one or more vertices of H results in a vertex-minimal graph $I \in \Pi'$. But if we ignore the same set of vertices in $H \in G[V(G) \setminus S]$, clearly, $I \in G[V(G) \setminus S]$. This contradicts our assumption that $G[V(G) \setminus S]$ is Π'-free. Therefore, there cannot be a $H \in \Pi$ such that $H \in G[V(G) \setminus S]$, which means that $G[V(G) \setminus S]$ is Π-free. □

We can make use of Lemma 4.19 to generalize the algorithms for H-free Deletion [VC]. We do, however, need to make the preconditions a little stronger. We want to make use of the vertex-minimal subset of Π, and so we require knowledge of its existence or we need to preprocess the graph class Π. Let us generalize Theorem 4.17 by repeatedly executing Algorithm 6 for different vertex-minimal $H \in \Pi$.

Theorem 4.20. *If Π is a graph property such that:*

(i) we have explicit knowledge of $\Pi' \subseteq \Pi$, which is the subset of q graphs that are vertex-minimal with respect to Π, and
(ii) there is a non-decreasing polynomial $p : \mathbb{N} \to \mathbb{N}$ such that all graphs $G \in \Pi'$ satisfy $|V(G)| \leq p(K)$, and
(iii) every graph in Π contains at least one edge,

then Π-free Deletion [VC] can be solved using $\mathcal{O}(q \cdot 2^K \cdot p(K)^K \cdot K! \cdot K \cdot p(K)! \cdot p(K)^2 \cdot n)$ time, where $K = |X|$, the size of the vertex cover.

Proof. This result can be achieved by either repeatedly applying Algorithm 6 on each of the q graphs in Π', or adjusting Algorithm 6 to search for each of the q graphs in Π' instead of only H. Then, using Theorem 4.17 and Lemma 4.19, the theorem follows. □

Note that we require explicit knowledge of Π' and $p(K)$ to achieve Theorem 4.20. Also note that we chose to write the $K!$ alternative here, as likely $K^{p(K)} > K!$ (i.e. for $p(K) = K$, a simple and small polynomial, we have that $K^{p(K)} = K^K > K!$).

Next, we look to further improve this upper bound, as its running time is not ideal. Note that so far, we have made no use at all of the characterization by few adjacencies of Π. However, this property is essential for Theorem 4.3 and Algorithm 2. Therefore, it is logical to pursue using this property in this setting as well. In this regard, we can consider the following lemma.

Lemma 4.21. *If Π is a graph property such that*

(i) every graph in Π is connected and contains at least one edge, and
(ii) Π is characterized by c_Π adjacencies,

and G is some graph with vertex cover X, $|X| = K$, and $S \subseteq V(G)$ some vertex set. Then $G[V(G) \setminus S]$ is Π-free if and only if $G[V(G) \setminus S]$ is Π'-free, where $\Pi' \subseteq \Pi$ contains only those graphs in Π with $\leq (c_\Pi + 1)K$ vertices.

Proof. Assume the preconditions in the lemma. When $G[V(G) \setminus S]$ is Π-free, it must also be Π'-free, as $\Pi' \subseteq \Pi$ by definition.

Now assume that $G[V(G) \setminus S]$ is not Π-free, let us say some $H \in \Pi$ occurs in the graph. Consider a vertex v of the graph H. Because Π is characterized by c_Π adjacencies, there exists a set $D \subseteq V(H)$ with $|D| \leq c_\Pi$ such that changing the adjacencies between v and $V(H) \setminus D$ does not change the presence of $H \in \Pi$. Remove all adjacencies existing between v and $V(H) \setminus D$. Then $\deg(v) \leq c_\Pi$. Our new version of H is still contained in Π, so we can repeat this process for every vertex in H. But then every vertex v in H has $\deg(v) \leq c_\Pi$. Given that H can contain at most K vertices of the vertex cover in G, each with degree at most c_Π, we know that this edited version of H can have at most $(c_\Pi + 1)K$ vertices. Although this version of H is still in Π, it might not exactly be in G, as we might have deleted essential adjacencies. However, changing all the adjacencies did not increase or decrease the number of vertices, as every graph in Π is connected and H remains in Π at every step. Therefore, every H occurring in $G[V(G) \setminus S]$ is in Π', and so the graph is not Π'-free either. □

In summary, Lemma 4.21 tells us that whenever every graph in Π is connected, and we already know the size of the given vertex cover, there may be graphs in Π that cannot occur in G simply because it would not fit with the vertex cover.

Let us motivate why we require every graph in Π to be connected for Lemma 4.21. If not every graph in Π is connected, the removal of adjacencies might leave a vertex without edges, but then this vertex might still be required for the presence in Π, which is problematic. Seeing that we want to bound the size of possible graphs in Π, to be able to bound the size of graphs we need that the vertex cover together with c_Π gives us information on the size of the graph, which is not the case for a disjoint union of graphs.

We can use Lemma 4.21 in combination with Theorem 4.20 to obtain a new result.

Theorem 4.22. *Given a graph G with vertex cover X, $|X| = K$, if Π is a graph property such that*

(i) every graph in Π is connected and contains at least one edge, and
(ii) Π is characterized by c_Π adjacencies, and
(iii) we have explicit knowledge of $\Pi' \subseteq \Pi$, which is the subset of q graphs of at most size $(c_\Pi + 1)K$ that are vertex-minimal with respect to Π,

then Π-FREE DELETION [VC] *can be solved using $\mathcal{O}(q \cdot 2^K \cdot ((c_\Pi+1)K)^K \cdot K! \cdot K \cdot ((c_\Pi+1)K)! \cdot ((c_\Pi+1)K)^2 \cdot n)$ time. Assuming $c_\Pi \geq 1$ this can be simplified to $\mathcal{O}(q \cdot 2^K \cdot {c_\Pi}^K \cdot K^{K+3} \cdot K! \cdot (c_\Pi K)! \cdot n)$ time.*

Proof. Given some Π characterized by c_Π adjacencies, where every graph in Π is connected, we can see that through Lemma 4.21 we only need to consider those graphs with size $\leq (c_\Pi + 1)K$ in Π, and with this subset, using Theorem 4.20 where $p(K) = (c_\Pi + 1)K$, the theorem follows. □

This result also applies to the streaming setting, as we can simply replace the application of Algorithm 6 in Theorem 4.20 with Algorithm 7. For completeness, we state the streaming equivalent of Theorem 4.22 as Theorem 4.23

Theorem 4.23. *Given a graph G with vertex cover X, $|X| = K$, if Π is a graph property such that*

(i) every graph in Π is connected and contains at least one edge, and
(ii) Π is characterized by c_Π adjacencies, and

(iii) we have explicit knowledge of $\Pi' \subseteq \Pi$*, which is the subset of* q *graphs of at most size* $(c_\Pi + 1)K$ *that are vertex-minimal with respect to* Π*,*

then Π-free Deletion [VC] *can be solved using* $\mathcal{O}(q \cdot 2^K \cdot c_\Pi^K \cdot K^{K+2} \cdot K! \cdot (c_\Pi K)!)$ *passes in the AL streaming model, using* $\widetilde{\mathcal{O}}((c_\Pi K)^2 + q \cdot (c_\Pi + 1)K)$ *space.*

Proof. See the proof of Theorem 4.22, where instead of applying Algorithm 6 we apply Algorithm 7. The factor $q \cdot (c_\Pi + 1)K$ in the memory usage comes from the fact that we need to explicitly store Π'. □

Now we have a result for Π-free Deletion [VC] in the streaming setting. However, the explicit knowledge of Π' gives us memory problems. That is, we have to store Π' somewhere to make this algorithm work, which takes $\widetilde{\mathcal{O}}(q \cdot (c_\Pi + 1)K)$ space. Note that q can range up to approximately $\mathcal{O}(K^K)$ if Π' contains many graphs. Regardless of the impact q has on the number of passes or running time, the purpose of the streaming setting is to optimize space usage, which can be disastrous if Π is chosen adversarially.

4.4 Π-free Deletion without explicit Π

One of the issues we have seen in the previous section is that having the graph property Π explicitly saved might cost us a lot of memory. To circumvent this, we can assume to be working with some *oracle*, which we can call to learn something about Π. In general, if we have an oracle algorithm $\mathcal{A}$, let us assume that it takes a graph G as input as a stream. We then denote $P_{\mathcal{A}}(n), M_{\mathcal{A}}(n)$ as respectively the number of passes and the memory use of the oracle algorithm $\mathcal{A}$ when called on a streamed (sub)graph G with n vertices. In this section, we will discuss two different oracles and their use for Π-free Deletion [VC]. In this entire section, we are concerned with the streaming model, and so our focus shall be on memory efficiency. We also assume that the graphs in Π have a maximum size ν, and that ν is known.

The first oracle model is where our oracle algorithm $\mathcal{A}_1$, when called on a graph G, returns whether or not $G \in \Pi$. The issue with using this oracle in comparison to Algorithm 7, is that in Algorithm 7 we rely on knowing whether a part of the vertex cover is contained in some $H \in \Pi$. This is not information the oracle can give us. Therefore, we need to approach solving the problem differently. The general idea is the following: we still branch on what part of the vertex cover is in the solution, and consider every subset of (the remaining part of) the vertex cover in each branch. To avoid having to test every combination of vertices outside the vertex cover together with this subset, we consider the notion of *equivalent vertices.* These are vertices with exactly the same set of edges towards the (total) vertex cover. More formally, two vertices $u, v \notin X$, where X is a vertex cover, are called *twins* when $N(u) = N(v)$, their neighbourhoods are equal. If EC is a set of vertices where each pair u, v are twins, and EC is maximal under this property, we call EC an *equivalence class.* When trying to test which graphs might be in Π, we can ignore twin vertices if we have tested one of them before. Also, if we delete a vertex from an equivalence class, we may need to delete the entire equivalence class. We will see the concept of twins appear again in Section 5.2.1. The benefit of using twins in the AL streaming model, is that we see all the necessary information, the adjacencies, local to each vertex. This means that we can find equivalence classes using few passes over the stream.

The main problem with this approach is that saving these twins (remembering the equivalence classes) is not memory efficient. Nonetheless, we use this idea here in Algorithm 8.

In Algorithm 8, the set EC saves the sizes of each equivalence class, and can be seen as a set of key-value pairs (key, val), where key is a K-bit string representing the adjacencies towards the vertex cover, and val is the number of vertices in this equivalence class. Notice that we can find the set EC using a single pass over the stream. This can be done by, for each vertex, locally saving its adjacencies as a K-bit string, and then finding and incrementing the correct counter in EC. As this process only requires information local to a single vertex, we need only one pass to do this for every vertex.

We use the notation $\textsc{Dict}_{EC_{\leq k}}$ to denote a dictionary ordering on choosing $\leq k$ 'vertices' out of the 2^K equivalence classes such that if an equivalence class *key* is chosen twice, then *val* is at least 2. This is essentially enumerating picking $\leq k$ vertices out of $V \setminus X$ except that we do not pick vertices explicitly, but we pick the equivalence classes they are from. An entry of this ordering is a set of key-value pairs $(key, count)$, such that *key* corresponds to some equivalence class, and *count* is the number of vertices we pick out of this equivalence class.

Algorithm 8 Π-FREE DELETION WITH $\mathcal{A}_1$(Graph $G = (V, E)$ in the AL model, integer ℓ, integer ν, Vertex Cover $X \subseteq V(G)$)

```
1: $S \leftarrow \textsc{First}(\mathcal{X}_{\leq \ell})$
2: while $S \in \mathcal{X}_{\leq \ell}, S \neq ♠$ do
3:     $Y \leftarrow X \setminus S$
4:     $S' \leftarrow S$
5:     if $\forall Y' \subseteq Y : \mathcal{A}_1(Y') =$ false then                ▷ Test if $Y$ is Π-free
6:         $EC \leftarrow$ Count the sizes of the equivalence classes of vertices in $V \setminus X$ with their
   adjacencies towards $Y$                                                    ▷ Use a pass for this
7:         if $\textsc{Search}(S', Y, EC)$ then return YES
8:     $S \leftarrow \textsc{Dict}_{\mathcal{X}_{\leq \ell}}(\textsc{Next}(S))$
9: return NO

10: function $\textsc{Search}$(solution set $S$, forbidden set $Y \subseteq X$, equivalence class sizes $EC$)
11:     $J \leftarrow \textsc{First}(\mathcal{Y}_{\leq \nu})$
12:     $I \leftarrow \textsc{First}(EC_{\leq(\nu - |J|)})$
13:     while $J \in \mathcal{Y}_{\leq \nu}, J \neq ♠$ do
14:         while $I \in EC_{\leq(\nu - |J|)}, I \neq ♠$ do
15:             if $\mathcal{A}_1(I, J)$ then                                ▷ $I \cup J \in \Pi$
16:                 if $|S| \geq \ell$ then return NO                      ▷ No budget to branch
17:                 else
18:                     for each $(k, count) \in I$ do
19:                         Update $EC$ such that $(k, val) \leftarrow (k, count - 1)$      ▷ Remove all but
   $count - 1$ vertices from class $k$
20:                         if $|S| + val - (count - 1) > \ell$ then return NO
21:                         else
22:                             $VS \leftarrow$ Find $val - (count - 1)$ vertices that belong to class $k$ ▷ Uses
   a pass
23:                             if $\textsc{Search}(S \cup VS, Y, EC)$ then return YES        ▷ Branch
24:             $I \leftarrow \textsc{Dict}_{EC_{\leq(\nu - |J|)}}(\textsc{Next}(I))$
25:         $J \leftarrow \textsc{Dict}_{\mathcal{Y}_{\leq \nu}}(\textsc{Next}(J))$
26:         $I \leftarrow \textsc{First}(EC_{\leq(\nu - |J|)})$
27:     return YES
```

It is good to notice that in Algorithm 8 we call $\mathcal{A}_1$ on sets I and J, while we previously stated it takes a stream as input. Therefore, each call to $\mathcal{A}_1$ actually consists of using a pass over the stream to select the sub-stream that corresponds to I and J (selecting vertices that correspond to the selected equivalence classes in I, and selecting the edges that are between these vertices and those in J) and pass this to $\mathcal{A}_1$. Actually, to be truly memory-less with this stream for $\mathcal{A}_1$ we have to do this same process every time $\mathcal{A}_1$ requires a pass over its input.

Theorem 4.24. *If Π is a graph property such that the maximum size of graphs in Π is ν, and $\mathcal{A}_1$ is an oracle algorithm that, when given subgraph H on h vertices, decides whether or not H is in Π using $P_{\mathcal{A}_1}(h)$ passes and $M_{\mathcal{A}_1}(h)$ bits of memory, then Algorithm 8 solves* Π-FREE DELETION [VC] *on a graph G on n vertices given as an AL stream with a vertex cover of size K using $\mathcal{O}(3^K \nu^{K+1} 2^{\nu K}[1 + P_{\mathcal{A}_1}(\nu)])$ passes and $\mathcal{O}(2^K(K + \log n) + \nu \log n + M_{\mathcal{A}_1}(\nu))$ bits of memory.*

Proof. Let us argue on the correctness of Algorithm 8. In essence, the algorithm tries every option of how an occurrence of a graph in Π can occur in G, by enumerating all subsets of the vertex cover (J) combined with vertices from outside the vertex cover (I). This process is optimized by use of the equivalence classes, removing multiple equivalent vertices at once to eliminate such an occurrence. If the algorithm returns YES, then for every combination of I and J, no occurrence of a graph in Π was found or branching happened on any such occurrence. Each branch eliminates the found occurrence of a graph in Π, as enough vertices are removed from an equivalence class to make the same occurrence impossible. As removing vertices from the graph cannot create new occurrences of graphs in Π, this means that every occurrence found was removed, and no new occurrences were created. But then there is no occurrence of a graph in Π, as every combination of such a graph occurring was tried. So the graph is Π-free. If the algorithm returns NO, then in every branch at some point we needed to delete vertices to remove an occurrence of a graph in Π but had not enough budget left. Removing any less than $val - (count - 1)$ vertices in line 22 results in at least *count* vertices of that equivalence class remaining, which means the same occurrence of a graph in Π persists. Therefore, the conclusion that we do not have enough budget is correct, and so there is no solution to the instance. We can conclude that the algorithm works correctly.

Let us analyse the space usage. Almost all sets used are bounded by K entries. The exceptions are the sets EC, I, and J. J contains at most ν vertices, and so uses at most $\mathcal{O}(\nu \log n)$ bits of space. EC contains at most 2^K key-value pairs, each using $K + \log n$ bits, so EC uses $\mathcal{O}(2^K(K + \log n))$ bits. The set I contains only subsets of EC. The oracle algorithm $\mathcal{A}_1$ uses at most $M_{\mathcal{A}_1}(\nu)$ bits. Therefore, the memory usage of this algorithm is $\mathcal{O}(2^K(K + \log n) + \nu \log n + M_{\mathcal{A}_1}(\nu))$ bits of space. Note that we branch on at most ν options together spanning at most ν vertices, which is a constant factor in the memory use (for remembering what we branch on when returning out of recursion). The algorithm also repeats the process for each I and J in each branch to avoid having to keep track of these sets in memory when returning from recursion.

It remains to show the number of passes used by the algorithm. Let it be clear that the number of passes made by the algorithm is dominated by the number of calls made to $\mathcal{A}_1$, which requires at least one pass. The number of calls made is heavily dependent on the total number of branches in the algorithm, together with how many options the sets I and J can span. This total can be concluded to be $\mathcal{O}(3^K \nu^{K+1} 2^{\nu K})$. Let us elaborate on this. The factor 3^K comes from the fact that any vertex in the vertex cover is either in J, in S, or in Y (and not in J). The factor ν^K comes from the worst case branching process, as we branch on at most ν different deletions, and we branch at most $\ell \leq K$ times. The remaining factor is the number of options we have for the set I, which picks, for each size i between 1 and ν, i times between at most 2^K options (the equivalence classes). This number of options is bounded by $\mathcal{O}(\nu 2^{\nu K})$. This means the total number of passes is bounded by $\mathcal{O}(3^K \nu^{K+1} 2^{\nu K}[1 + P_{\mathcal{A}_1}(\nu)])$. □

Let us note that the memory use of Algorithm 8 is far from ideal. The aim with the oracle approach was to save more memory by attempting to avoid saving Π explicitly, but now we have a memory complexity with a factor $K2^K$, which might well be more than saving Π explicitly. To remedy this, next we look at a different oracle model, which will actually also lead to a more memory efficient algorithm for oracle $\mathcal{A}_1$ as well.

The second oracle model is where our oracle algorithm $\mathcal{A}_2$, when called on a graph G, returns whether or not G is induced Π-free (that is, it returns YES when it is induced Π-free, and NO if there is an occurrence of a graph in Π in G). The advantage of this model is that we can be less exact in our calls, namely by always passing the entire vertex cover (that is, the part that is not in the solution) together with some set of vertices. This avoids some extra enumerations. The idea for the algorithm with this oracle is the following: we call on the oracle with small to increasingly large sets, such that whenever we get returned that this set is not Π-free, we can delete a single vertex to make it Π-free. This can be done by first calling the oracle on the vertex cover itself, and then on the vertex cover together with a single vertex for every vertex outside the vertex cover, then with two, etc.. The problem with this method is the complexity in n that will be present, but as we know the maximum size of graphs in Π, ν, this is still a bounded polynomial.

Algorithm 9 Π-FREE DELETION WITH $\mathcal{A}_2$(Graph $G = (V, E)$ in the streaming model, integer ℓ, integer ν, Vertex Cover $X \subseteq V(G)$)

```
1: $S \leftarrow$ First($\mathcal{X}_{\leq \ell}$)
2: while $S \in \mathcal{X}_{\leq \ell}, S \neq ♠$ do
3:     $Y \leftarrow X \setminus S$
4:     $S' \leftarrow S$
5:     if $\mathcal{A}_2(Y)$ then
6:         if Search($S'$, $Y$, $\emptyset$) then return YES
7:     $S \leftarrow$ Dict$_{\mathcal{X}_{\leq \ell}}$(Next($S$))
8: return NO

9: function Search(solution set $S$, forbidden set $Y \subseteq X$, set $I$)
10:     if $I = \emptyset$ then $I \leftarrow$ First($(V \setminus X)_{\leq \nu}$)
11:     else
12:         $I \leftarrow$ Dict$_{(V \setminus X)_{\leq \nu}}$(Next($I$))
13:     if $I = ♠$ then return YES
14:     else if $I \cap S \neq \emptyset$ then return Search($S$, $Y$, $I$)        ▷ This $I$ is invalid
15:     else if $\mathcal{A}_2(Y \cup I)$ then return Search($S$, $Y$, $I$)        ▷ $Y \cup I$ is Π-free
16:     else if $|S| = \ell$ then return NO        ▷ No budget to branch
17:     else
18:         for each $v \in I$ do
19:             if Search($S \cup \{v\}$, $Y$, $I$) then return YES    ▷ Branch, this makes $Y \cup I$ Π-free
20:     return NO
```

Let us shortly elaborate on some details of Algorithm 9. Notice that the algorithm calls on the oracle algorithm $\mathcal{A}_2$ using only a vertex set V, while we previously stated that the oracle accepts a stream as input. Once again, this difference can be solved by using a pass over the stream, and giving only those edges between the vertices of V to the oracle. Actually, to be truly memory-less with this stream for $\mathcal{A}_2$ we have to do this same process every time $\mathcal{A}_2$ requires a pass over its input. Therefore, each call to the oracle algorithm requires a pass over the stream, plus a pass for the every time the oracle uses a pass over its input.

We also note that Algorithm 9 is similar to the Π-FREE MODIFICATION algorithm of Cai [13].

We are now ready to prove the complexity of Algorithm 9. Notice that we do not explicitly state what streaming model we use, as this is dependent on the type of stream the oracle algorithm accepts. So if the oracle algorithm works on a streaming model X, then so does Algorithm 9.

Theorem 4.25. *If* Π *is a graph property such that the maximum size of graphs in* Π *is* ν*, and* $\mathcal{A}_2$ *is an oracle algorithm that, when given subgraph* H *on* h *vertices, decides whether or not* H *is* Π*-free using* $P_{\mathcal{A}_2}(h)$ *passes and* $M_{\mathcal{A}_2}(h)$ *bits of memory, then Algorithm 9 solves* Π-FREE DELETION [VC] *on a graph* G *on* n *vertices given as a stream with a vertex cover of size* K *using* $\mathcal{O}(2^K\nu^{K+1}n^\nu[1+P_{\mathcal{A}_2}(K+\nu)])$ *passes and* $\mathcal{O}(\nu K \cdot \log n + M_{\mathcal{A}_2}(K+\nu))$ *bits of memory.*

Proof. The correctness of the algorithm can be seen as follows. Whenever the algorithm finds a subset of the graph which is not induced Π-free, it branches on one of the possible vertex deletions. This always makes this subset Π-free, as in previous iterations every subset with one vertex less was tried, to which the oracle told us that it was Π-free (otherwise, we would have branched there already, and this subset could not occur in the current iteration). Therefore, the branching process removes the found occurrences of induced Π graphs. The algorithm only returns YES when all subsets of at most ν vertices have been tried as I, and as every graph in Π has at most ν vertices, this includes every possibility of a graph in Π appearing in G. Therefore, if at this point we have a set S of at most ℓ vertices, then $G[V \setminus S]$ must be Π-free, and so S is a solution to Π-FREE DELETION [VC]. If the algorithm returns NO, then for every possible subset of the vertex cover contained in S, there is not enough budget to remove all occurrences of induced Π graphs, and therefore there is no solution to Π-FREE DELETION [VC]. We can conclude the algorithm works correctly.

Let us analyse the space usage. The algorithm requires space for the sets S, S', Y, X, I, and for the oracle algorithm $\mathcal{A}_2$. The size of each of the sets S, S', Y, and X is bounded by K entries, so we use $\mathcal{O}(K \log n)$ bits memory for them. The set I contains at most ν entries, and therefore requires at most $\mathcal{O}(\nu \log n)$ bits of space. The oracle algorithm is called on graphs of at most $K+\nu$ vertices, and so requires $M_{\mathcal{A}_2}(K+\nu)$ bits of memory. To handle branching correctly, we need to save the set I and all vertices we branch on every time we branch. These sets have a maximum size of ν, and the search tree has a maximum depth of K, so this takes $\mathcal{O}(\nu K \log n)$ bits of memory. Therefore, the total memory usage comes down to $\mathcal{O}(\nu K \cdot \log n + M_{\mathcal{A}_2}(K+\nu))$ bits. Seeing as ν is a constant, we could further simplify this, but we choose to be more explicit here.

As mentioned, we require a pass over the stream to call on the oracle. Therefore, the number of passes on the stream is dependent on the number of calls to the oracle algorithm $\mathcal{A}_2$. In the worst case, we call on $\mathcal{A}_2$ in every branch for every set I. The number of branches is bounded by $\mathcal{O}(2^K\nu^K)$, as this encompasses every subset of the vertex cover and every deletion branch is on at most ν vertices. There are $\sum_{i=1}^{\nu}\binom{n}{i} = \mathcal{O}(\nu n^\nu)$ different subsets I. This means the total number of calls to $\mathcal{A}_2$ is bounded by $\mathcal{O}(2^K\nu^{K+1}n^\nu)$, which means the total number of passes is bounded by $\mathcal{O}(2^K\nu^{K+1}n^\nu[1+P_{\mathcal{A}_2}(K+\nu)])$. □

Algorithm 9 can also be executed using oracle algorithm $\mathcal{A}_1$, but instead of passing the entire vertex cover each time, we would need to do this for every subset of the vertex cover. This would make the 2^K factor in the number of passes a 3^K factor (a vertex in the vertex cover can either not be in Y, or be in Y and not passed, or in Y and passed to $\mathcal{A}_1$). Let us shortly formalize this in a theorem.

Theorem 4.26. *If* Π *is a graph property such that the maximum size of graphs in* Π *is* ν*, and* $\mathcal{A}_1$ *is an oracle algorithm that, when given subgraph* H *on* h *vertices, decides whether or not* H

is in Π *using* $P_{\mathcal{A}_1}(h)$ *passes and* $M_{\mathcal{A}_1}(h)$ *bits of memory, then Algorithm 9 can be adapted to solve* Π-free Deletion [VC] *on a graph* G *on* n *vertices given as a stream with a vertex cover of size* K *using* $\mathcal{O}(3^K \nu^{K+1} n^\nu [1 + P_{\mathcal{A}_1}(\nu)])$ *passes and* $\mathcal{O}(\nu K \cdot \log n + M_{\mathcal{A}_1}(\nu))$ *bits of memory.*

Proof. The only difference between oracle algorithm $\mathcal{A}_1$ and $\mathcal{A}_2$ is that $\mathcal{A}_1$ requires an exact occurrence to be input, while for oracle $\mathcal{A}_2$ a larger set can suffice. As Algorithm 9 already enumerates all of the possibilities for vertices outside the vertex cover, only refinement is necessary for how the vertex cover itself is passed to the oracle. At each location where a call happens to $\mathcal{A}_2$ in Algorithm 9, we actually want to replace this by an enumeration of calls for each subset of the vertex cover, similar to how Algorithm 8 has a combination of sets I and J. By the correctness of Algorithm 9, this approach also leads to a correct solution. The number of calls to $\mathcal{A}_1$ does increase in comparison to $\mathcal{A}_2$, however. We can include the increase in complexity in the already existing branching complexity on the vertex cover. In any branch, any vertex in the vertex cover is either in S, or in Y and passed to $\mathcal{A}_1$, or in Y and not passed. This means that the number of calls to $\mathcal{A}_1$ is bounded by $\mathcal{O}(3^K \nu^{K+1} n^\nu)$, and therefore the total number of passes is bounded by $\mathcal{O}(3^K \nu^{K+1} n^\nu [1 + P_{\mathcal{A}_1}(\nu)])$. The memory usage of the algorithm is asymptotically the same as that of Algorithm 9, with $\mathcal{A}_2$ replaced by $\mathcal{A}_1$. □

We have now seen some algorithms for using an oracle to obtain information about Π instead of explicitly saving it. We do demand the maximum size for a graph in Π to be some given constant ν, which is hopefully small. This way, we have gotten closer to a memory-optimal algorithm, but the number of passes over the stream has increased significantly in comparison to earlier algorithms.

4.5 Odd Cycle Transversal

Now that we have seen results for the general Π-free Deletion [VC], we can wonder whether there are specific scenarios where we can significantly improve on the number of passes. One of these cases is for the problem of Odd Cycle Transversal. The interest in this problem comes from the FPT algorithm using iterative compression provided in [23, Section 4.4], based on work by Reed *et al.* [57]. Although Chitnis and Cormode [17] have shown how iterative compression can be used in the streaming setting, adapting the algorithm out of [23, Section 4.4] is difficult. The main cause for this is the use of a maximum-flow algorithm to compute a minimum cut, which does not lend itself well to the streaming setting because of its memory requirements. Nonetheless, this algorithm inspired us to think on an efficient algorithm for Odd Cycle Transversal in the streaming setting, using once again vertex cover as a parameter.

Let us formally introduce the problem of Odd Cycle Transversal [VC] (OCT).

Odd Cycle Transversal [VC]
Input: A graph G with a vertex cover X, and an integer ℓ.
Parameter: The size $K := |X|$ of the vertex cover.
Question: Is there a set $S \subseteq V(G)$ of size at most ℓ such that $G[V(G) \setminus S]$ contains no induced odd cycles?

It is well known that a graph without odd cycles is a bipartite graph and vice versa. A useful property in this regard is that a bipartite graph $G = (V, E)$ always admits a proper 2-colouring, that is, a partition of V into two sets V_1 and V_2 such that for every edge $(u, v) \in E$ we have that $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$ (u and v are not in the same set).

The idea for the streaming algorithm for OCT is simple. We once again guess what part of the vertex cover is in the solution, and then we guess the colouring of the remaining part of the vertex cover. If we do this, we can deterministically determine which other vertices outside the vertex cover need to be deleted. This last step can be done in one pass if we use the AL streaming model. We formally give this algorithm as Algorithm 10.

Algorithm 10 OCT(Graph $G = (V, E)$ in the AL model, integer ℓ, Vertex Cover $X \subseteq V(G)$)

```
1:  $S \leftarrow \textsc{First}(\mathcal{X}_{\leq \ell})$
2:  while $S \in \mathcal{X}_{\leq \ell}, S \neq ♠$ do
3:      $Y \leftarrow X \setminus S$
4:      $Y_1 \leftarrow \textsc{First}(\mathcal{Y})$
5:      $Y_2 \leftarrow Y \setminus Y_1$
6:      while $Y_1 \subseteq Y, Y_1 \neq ♠$ do
7:          $success \leftarrow true, S' \leftarrow S$                ▷ Reset local values
8:          for each $v \in V \setminus S'$ do                      ▷ Use one pass
9:              if $v \in Y$ and $v \in Y_i$ then                  ▷ $v \in X$
10:                 Check that all neighbours of $v$ in $Y$ are in $Y_{3-i}$
11:                 If one is not, $success \leftarrow false$
12:             else                                               ▷ $v \notin X$
13:                 Check if all neighbours of $v$ in $Y$ are in the same $Y_i$
14:                 If not, and $|S'| < \ell$, $S' \leftarrow S' \cup \{v\}$
15:                 Else, $success \leftarrow false$
16:         if $|S'| \leq \ell$ and $success$ then return YES
17:         $Y_1 \leftarrow \textsc{Dict}_{\mathcal{Y}}(\textsc{Next}(Y_1))$          ▷ Try the next colouring
18:         $Y_2 \leftarrow Y \setminus Y_1$
19:     $S \leftarrow \textsc{Dict}_{\mathcal{X}_{\leq \ell}}(\textsc{Next}(S))$
20: return NO
```

Let us formally show the performance of Algorithm 10.

Theorem 4.27. *Given a graph G given as an AL stream with vertex cover X, $|X| = K$, Algorithm 10 solves* Odd Cycle Transversal [VC] *using $\mathcal{O}(3^K)$ passes and $\mathcal{O}(K \log n)$ bits of memory.*

Proof. Let us first prove the correctness of Algorithm 10. Let $G = (V, E)$ be a graph with vertex cover X, $|X| = K$. Let O, $|O| \leq \ell$ be a solution for Odd Cycle Transversal [VC]. Denote $Y' = X \cap O$ as the part of O that is contained in the vertex cover. Because Algorithm 10 enumerates all possibilities for the set Y, at some point it must consider $Y = Y'$. As O is a solution, $G[X \setminus Y']$ must be bipartite, and so admits a proper 2-colouring Y_1', Y_2'. For $Y = Y'$, Algorithm 10 considers at some point $Y_1 = Y_1'$ and $Y_2 = Y_2'$ ($Y_1' \cup Y_2' = Y' = Y$ because Y_1' and Y_2' form a proper 2 colouring), because it considers all possibilities for the set $Y_1 \subseteq Y$. As Y_1 and Y_2 now form a proper 2-colouring of Y, the check in line 10 never fails. The check in line 13 only fails if a vertex outside of X is adjacent to two different coloured vertices in Y. But then this vertex must also be in O, as Y_1 and Y_2 mimic exactly the 2-colouring in $G[X \setminus Y']$. Therefore, Algorithm 10 can at least find O as well, which means it returns YES. For the reverse implication, when Algorithm 10 returns YES, it has found a set S such that $|S| \leq \ell$ and $G[V \setminus S]$ admits a proper 2-colouring given by deterministically adding vertices outside the vertex cover to Y_1 and Y_2. But then S is a solution to Odd Cycle Transversal [VC]. We can conclude that Algorithm 10 works correctly.

Let us analyse the memory usage of Algorithm 10. All sets used in the algorithm have size at most K ($\ell \leq K$). The only worry is whether or not the checks in lines 10 and 13 require more memory. The first check only requires us to remember in what set the vertex v is contained in, and whether or not we have seen a 'wrong' colour yet, which should only take a constant number of bits. The second check merely needs to remember in what set all neighbours up until this point were, which should also only take a constant number of bits. Therefore, the algorithm uses $\mathcal{O}(K \log n)$ bits of memory.

Let us analyse the number of passes of Algorithm 10 does over the stream. Firstly, the for-loop over all vertices only requires a single pass because the checks only need to know what all the neighbours of the current vertex are, which is what the AL stream gives us. The total number of times this for-loop can be executed is bounded by $\mathcal{O}(3^K)$, as any vertex in the vertex cover can either be in S, in Y_1 or in Y_2. We can conclude that Algorithm 10 uses $\mathcal{O}(3^K)$ passes over the graph stream. □

We can ask ourselves if we can do better. After all, finding a 2-colouring can be quite deterministic, while in Algorithm 10 we enumerate every possibility for a 2-colouring, many of which are invalid. After branching on what part of the vertex cover is in the solution, we can attempt to find a 2-colouring on the remainder of the vertex cover. This can be done by giving some vertex a colour 1, and giving all its neighbours the colour 2, all their neighbours colour 1, etc.. This process colours the entire vertex cover if it is connected. However, problems arise if it is not. If there are a few connected components in the vertex cover, connected with vertices outside the vertex cover, it can be difficult to decide which of these vertices should enforce a relation in colouring, and which should be deleted. To avoid this complication, we can enumerate all combinations of colourings on the different components of the vertex cover. That is, each component has two unique ways of being 2-coloured (if it allows a 2-colouring at all), one by the aforementioned expansion process, and the exact complement colouring. If each component allows two 2-colourings, we can enumerate the combinations of what colouring each component picks. For every such combination, we can do a simple pass over all vertices outside the vertex cover to find whether or not this leads to a solution.

To make the process of finding the 2-colourings in the components of the vertex cover easier in the streaming setting, we might want to save the entire vertex cover (with edges) in memory. This does indeed increase the memory complexity, but it would mean that the worst case complexity for the number of passes is still equivalent to that of Algorithm 10. Notice that the worst case complexity might be the same, but in many cases, this algorithm can perform better, as the number of components in the vertex cover might be a lot smaller than the number of vertices in the vertex cover.

We give some pseudocode for this approach, see Algorithm 11.

Theorem 4.28. *Given a graph G given as an AL stream with vertex cover X, $|X| = K$, Algorithm 11 solves* Odd Cycle Transversal [VC] *using $\mathcal{O}(3^K)$ passes and $\mathcal{O}(K^2 \log n)$ bits of memory.*

Proof. The correctness of Algorithm 11 quickly follows from the correctness of Algorithm 10. Where Algorithm 10 enumerates all 2-colourings of Y, Algorithm 11 only enumerates those which are feasible 2-colourings for Y (combinations of 2-coloured components). This means Algorithm 11 only leaves out colourings which are not feasible anyway, which means that it still works correctly.

The number of passes is heavily dependent on the amount of connected components in $G[Y]$ in an iteration. But, in worst case, this is $|Y|$, which would mean it considers all 2-colourings of Y. But this is a worst case where the behaviour exactly mimics that of Algorithm 10, and so the worst case number of passes is the same.

Algorithm 11 OCT-CC(Graph $G = (V, E)$ in the AL model, integer ℓ, Vertex Cover $X \subseteq V(G)$)

```
1: $S \leftarrow \textsc{First}(\mathcal{X}_{\leq \ell})$
2: while $S \in \mathcal{X}_{\leq \ell}, S \neq ♠$ do
3:     $Y \leftarrow X \setminus S$
4:     Use a pass to find and save the edges in $Y$
5:     Find the connected components and their two 2-colourings
6:     If this fails, move to the next option for $S$
7:     for each Combination of colourings of the CCs do        ▷ Does $2^{\#\text{CCs}}$ iterations
8:         $success \leftarrow true, S' \leftarrow S$                  ▷ Reset local values
9:         for each $v \in V \setminus (X \cup S')$ do                   ▷ Use one pass
10:            Check if all neighbours of $v$ in $Y$ have the same colour
11:            If not, and $|S'| < \ell$, $S' \leftarrow S' \cup \{v\}$
12:            Else, $success \leftarrow false$
13:        if $|S'| \leq \ell$ and $success$ then return YES
14:    $S \leftarrow \textsc{Dict}_{\mathcal{X}_{\leq \ell}}(\textsc{Next}(S))$
15: return NO
```

Let it be clear that the memory use is $\mathcal{O}(K^2 \log n)$ bits, as we save (a part of) the vertex cover with edges to enable easy colouring and component finding. The rest of the sets in the algorithm use $\mathcal{O}(K \log n)$ bits of memory. □

Let us denote again that Algorithm 11 on paper is strictly worse than Algorithm 10 in worst case complexity. However, we believe there to be many cases where Algorithm 11 can outperform Algorithm 10 because of its behaviour of clever enumeration instead of trying all possibilities.

4.6 Lower Bounds

With all the positive results, we can ask ourselves if there are lower bounds for Π-free Deletion problems as well. To prove such lower bounds, commonly, reductions are done from problems in communication complexity. These are problems where two players, Alice and Bob, get some input and want to learn some information using as little communication as possible. An example of such a problem is Disjointness, which we will use, and it is defined as follows.

Disjointness
Input: Alice has a string $x \in \{0,1\}^n$ given by $x_1 x_2 \dots x_n$. Bob has a string $y \in \{0,1\}^n$ given by $y_1 y_2 \dots y_n$.
Question: Bob wants to check if $\exists 1 \leq i \leq n$ such that $x_i = y_i = 1$. (Formally, the answer is NO if this is the case.)

To prove lower bounds, it is useful to give a proposition that provides us with reduction results. The following proposition is given and used by Bishnu *et al.* [8], and gives us one important consequence of reductions from a problem in communication complexity to a problem for streaming algorithms:

Proposition 4.29. *(Rephrasing of item (ii) of [8, Proposition 5.6]) If we can show a reduction from* Disjointness *to problem* Π *in streaming model* $\mathcal{M}$ *such that the reduction uses a 1-pass streaming algorithm of* Π *as a subroutine, then any streaming algorithm working in the model* $\mathcal{M}$ *for* Π *that uses* p *passes requires* $\Omega(n/p)$ *bits of memory, for any* $p \in \mathbb{N}$ *[2, 9, 19].*

Proposition 4.29 allows us to prove lower bounds for problems in the streaming setting by performing a reduction. These structure of these reductions is relatively simple, have Alice and Bob construct the input for a streaming algorithm depending on their input to Disjointness. If we do this in such a manner that the solution the streaming algorithm outputs gives us exactly the answer to Disjointness, we can conclude that the streaming algorithm must abide the lower bound of Disjointness. This is essentially what Proposition 4.29 entails, and we will use this our proofs.

4.6.1 Π-free Deletion

Chitnis *et al.* [18] give a lower bound proof for Π-free Deletion under some conditions of Π, that is, when Π is *bad*.

Definition 4.30. ([18, Definition 6.3]) Let Π be a set of graphs such that each graph in Π is connected. We say that Π is *bad* if there is a graph $H \in \Pi$ such that

- H is a minimal element of Π under the operation of taking subgraphs, i.e., no proper subgraph of H is in Π, and
- H has at least two distinct edges.

With a reduction from Disjointness, Chitnis *et al.* prove the following theorem.

Theorem 4.31. *([18, Theorem 6.3]) For a bad set of graphs* Π*, any p-pass (randomized) streaming algorithm in the EA streaming model for the* Π-free Deletion [ℓ] *problem needs* $\Omega(n/p)$ *bits of space.*

As mentioned, Chitnis *et al.* prove Theorem 4.31 by giving a reduction from Disjointness, which consists of showing a construction where Alice and Bob use a streaming algorithm for Π-free Deletion to solve Disjointness. However, what Chitnis *et al.* do not describe, is how exactly Alice and Bob give their 'input' as a stream to the algorithm for Π-free Deletion. Therefore, this proof is natively for the EA streaming model (and so also for the DEA streaming model), as this model requires no structure on the input stream. However, if we specify how Alice and Bob provide their input to the streaming algorithm, we can extend the proof to other streaming models.

Theorem 4.32. *For a bad set of graphs* Π*, any p-pass (randomized) streaming algorithm in the VA streaming model for the* Π-free Deletion [ℓ] *problem needs* $\Omega(n/p)$ *bits of space.*

Proof. We add onto the proof of [18, Theorem 6.3], by specifying how Alice and Bob provide the input to the p-pass streaming algorithm.

Let H be the minimal graph in Π which has at least two distinct edges, say e_1 and e_2. Let $H' := H \setminus \{e_1, e_2\}$. As a reminder, Chitnis *et al.* create as an input for the streaming algorithm n copies of H', where in copy i we add the edges e_1 and e_2 if and only if the input of Disjointness has a 1 for index i for Alice and Bob respectively.

As e_1 and e_2 are distinct, there must be some vertex on which e_2 is incident, while e_1 is not. Call this vertex v. For every pass the algorithm requires, we do the following. We provide all the copies of H as input to the streaming algorithm by letting Alice input all vertices $V(H) \setminus \{v\}$ as a VA stream. Note that Alice has enough information to do this, as the edge e_2 in each copy of H is never included in this part of the stream. Then Alice passes the memory of the streaming algorithm to Bob, who inputs the edges adjacent to the vertex v for each copy of H (which includes e_2 if and only if the respective bit in the input of Disjointness is 1). This ends a pass of the stream for the streaming algorithm.

Note that Alice and Bob have input the exact specification of a graph as described by Chitnis *et al.*, but now as a VA stream. The correctness of the reduction follows from the proof given by Chitnis *et al.* [18, Theorem 6.3]. Hence, the theorem follows. □

Now we would like to do the same for the AL streaming model. However, difficulty comes in when the two distinct edges e_1 and e_2 have an overlapping endpoint, as neither Alice nor Bob have enough information to handle inputting the adjacencies for this vertex completely. Therefore, we need an extra demand on the graph $H \in \Pi$, that not only the two edges are distinct, but also disjoint. Let us formalize this in a theorem.

Theorem 4.33. *For a set of graphs* Π *such that each graph in* Π *is connected, and there is a graph* $H \in \Pi$ *such that*

- H *is a minimal element of* Π *under the operation of taking subgraphs, i.e., no proper subgraph of* H *is in* Π*, and*
- H *has at least two disjoint edges,*

then any p*-pass (randomized) streaming algorithm working on the AL streaming model for* Π-FREE DELETION $[k]$ *needs* $\Omega(n/p)$ *bits of space.*

Proof. We add onto the proof of [18, Theorem 6.3], by specifying how Alice and Bob provide the input to the p-pass streaming algorithm.

Let H be the minimal graph in Π which has at least two disjoint edges, say e_1 and e_2. Let $H' := H \setminus \{e_1, e_2\}$. As a reminder, Chitnis *et al.* create as an input for the streaming algorithm n copies of H', where in copy i we add the edges e_1 and e_2 if and only if the input of DISJOINTNESS has a 1 for index i for Alice and Bob respectively.

As e_1 and e_2 are disjoint, e_2 is incident on two vertices v, w which are not incident to e_1. For every pass the algorithm requires, we do the following. We provide all the copies of H as input to the streaming algorithm by letting Alice input all vertices $V(H) \setminus \{v, w\}$ as an AL stream. Note that Alice has enough information to do this, as the vertices incident on the edge e_2 in each copy of H is never included in this part of the stream. Then Alice passes the memory of the streaming algorithm to Bob, who inputs the edges incident to the vertices v, w for each copy of H (which includes e_2 if and only if the respective bit in the input of DISJOINTNESS is 1). This ends a pass of the stream for the streaming algorithm.

Note that Alice and Bob have input the exact specification of a graph as described by Chitnis *et al.*, but now as a AL stream. The correctness of the reduction follows from the proof given by Chitnis *et al.* [18, Theorem 6.3]. Hence, the theorem follows. □

Examples for which Theorem 4.33 provides a lower bound include EVEN CYCLE TRANSVERSAL $[k]$ (where $\Pi = \{C_4, C_6, \ldots\}$), and ODD CYCLE TRANSVERSAL $[k]$ (where $\Pi = \{C_3, C_5, C_7, \ldots\}$ and we can pick $H = C_5$). This means it also includes FEEDBACK VERTEX SET $[k]$, where Π is the set of all cycles, and we can pick C_4 as H.

Notice that in the above theorems the construction we make has a vertex cover size linear in n. Therefore, it might be interesting to consider if problems are hard even when the size of a minimum vertex cover is bounded, as many of our algorithms use a bounded vertex cover size.

4.6.2 *H*-free Deletion with bounded vertex cover size

It is interesting to pursue lower bounds for H-FREE DELETION, as Π-FREE DELETION is a generalization of this problem, and so lower bounds for H-FREE DELETION are also lower bounds for the more general version of the problem. It is also interesting to pursue lower bounds for

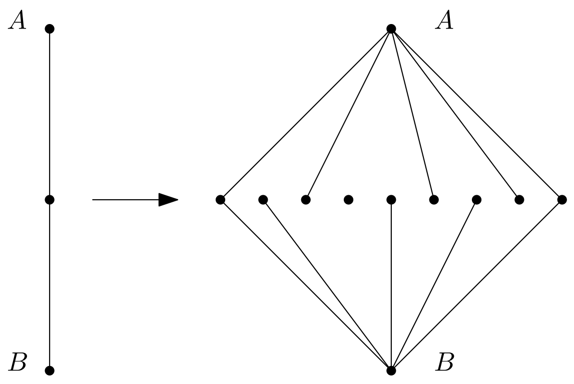

Figure 3: A reduction technique which we call a *double fan.* The right construction can imitate the input to the Disjointness instance, forming the left construction if and only if the answer to Disjointness is NO.

this problem parameterized by vertex cover, as we extensively use this parameter in our upper bounds.

In the next few proofs, we will use a construction similar to that used by Bishnu *et al.* [8] to find lower bounds for H-free Deletion with bounded vertex cover size. In essence, we reduce from the communication complexity problem Disjointness to H-free Deletion in the following manner. We take a copy of H, and extend some path of three vertices to the construction in Figure 3, which we will call a *double fan.* The n vertices in the overlap of the two fans we will call the *center vertices.* The idea of this construction is that both Alice and Bob input the edges of one of the fans in the double fan, as those edges will be determined by the input to Disjointness. If there is some $1 \leq i \leq n$ such that the i-th bit in both Alice's and Bobs input is 1, then the double fan will have a completed path from A to B, creating an induced copy of H in the graph. If we make sure the budget is $\ell = 0$, then the answer to H-free Deletion must be NO if and only if the answer to Disjointness is NO. If we manage this, Proposition 4.29 gives us a lower bound result.

It is important to note that this construction does not always make a correct reduction. Most importantly, it innately has trouble if H is some star. This is because the double fan construction can form exactly H even if Bobs input exists entirely of zeroes. Therefore, we must be careful to form conditions that exclude stars from the lower bounds proofs.

Theorem 4.34. *If H is a connected graph with at least 3 edges and a vertex of degree 2, then any algorithm for solving H-*free Deletion *[VC] on a graph G with $K \geq |\text{VC}(H)|+1$ requires at least n/p bits when using p passes in the VA/EA models, even when the solution size $\ell = 0$.*

Proof. Similar to the reductions given by Bishnu *et al.* [8], we give a reduction from Disjointness to H-free Deletion [VC] in the VA model when the solution size parameter $k = 0$. The idea is to build a graph G with bounded vertex cover size, and construct edges according to the input of Disjointness, such that G is H-free if and only if the output of Disjointness is YES.

Let $\mathcal{A}$ be a one-pass streaming algorithm that solves H-free Deletion $[K]$ in the VA model, such that $|\text{VC}(G)| \leq K \leq |\text{VC}(H)| + 1$, and the space used is $o(n)$. Let G be a graph with $n + |V(H)| - 1$ vertices consisting of H where a degree-2 vertex in H is expanded to a double fan (i.e., the two adjacencies of this degree-2 vertex correspond to A and B, and the degree-2 vertex is replaced by the n center vertices of the double fan). Let x, y be the input strings, consisting of n bits each, of Alice and Bob for Disjointness, respectively.

Alice exposes to $\mathcal{A}$ all the vertices of G, except for the vertex B. Here, Alice exposes an edge between A and the i-th center vertex of the double fan if and only if the i-th bit of x is 1. Notice that Alice can expose all these vertices according to the VA model, as only the addition of the vertex B will require information of the input of Bob, y. If Alice has exposed all vertices of G except for B, then she passes the memory of $\mathcal{A}$ to Bob. Bob then exposes the vertex B, including an edge between B and the i-th center vertex of the double fan if and only if the i-th bit of y is 1. This completes the input to $\mathcal{A}$.

From the construction, observe that $|VC(G)| \leq K \leq |VC(H)|+1$, as we may need to include both A and B in the vertex cover in G while it was optimal to include the degree-2 vertex in the vertex cover in H.

If the answer to DISJOINTNESS is NO, that is, there exists an index i such that $x_i = y_i = 1$, then in G the edges from A and B connect in the i-th center vertex, creating an induced copy of H in G, and so the graph is not H-free. Because $k = 0$, H-FREE DELETION $[K]$ must also be answered with NO.

If the answer to DISJOINTNESS is YES, that is, there is no index i such that $x_i = y_i = 1$, then there is no path from A to B through a center vertex in G. We will show that there is no induced occurrence of H in G. If the degree-2 vertex that we split into the double fan is contained in a cycle in H, then now this cycle is no longer present in the graph. As the rest of the graph simply consists of a (partial) copy of H, this means there cannot be enough cycles in the graph to get exactly H, and so H does not appear in G. Otherwise, the degree-2 vertex is not contained in a cycle, and so there is no path between A and B. As H has at least three edges, there must be some edge incident to A or B that is not incident to a center vertex. Consider in H the longest path that this degree-2 vertex is contained in. This path must have at least three edges and contain both A and B, as H is connected. However, because there is no path between A and B, the longest path in G containing A or B must be smaller that the longest path in H containing A and B. Hence, the only way for H to occur in G is for this path through A and B to occur somewhere else in G. However, this would mean there is now some other path of at least the same length that needs 'another place', as it were, to make the induced copy of H appear in G. We can see that repeating this process always yields in another path of at least the same length which needs to occur in G. However, since we destroyed at least one path of at least this length, all these paths cannot appear in G. Hence, the answer to H-FREE DELETION [VC] is YES.

Now, from Proposition 4.29 it follows that any algorithm for solving H-FREE DELETION [VC] on a graph G with $K \leq |\mathrm{VC}(H)|+1$ requires at least n/p bits when using p passes in the VA/EA models, even when $\ell = 0$. This can be generalized for every ℓ by adding ℓ disjoint copies of H to G, which also increases the vertex cover of G by a constant amount for each copy. □

There are other conditions for which this reduction works as well.

Theorem 4.35. *If H is a graph with no vertex of degree 1, then any algorithm for solving H-FREE DELETION [VC] on a graph G with $K \geq |V(H)|$ requires at least n/p bits when using p passes in the VA/EA models, even when the solution size $\ell = 0$.*

Proof. Similar to the reductions given by Bishnu *et al.* [8], we give a reduction from DISJOINTNESS to H-FREE DELETION [VC] in the VA model when the solution size parameter $k = 0$. The idea is to build a graph G with bounded vertex cover size, and construct edges according to the input of DISJOINTNESS, such that G is H-free if and only if the output of DISJOINTNESS is YES.

Let $\mathcal{A}$ be a one-pass streaming algorithm that solves H-FREE DELETION [VC] in the VA model, such that $|\mathrm{VC}(G)| \leq K \leq |V(H)|$, and the space used is $o(n)$. Let G be a graph with $n + |V(H)| - 1$ vertices consisting of H where a vertex of minimal degree in H is expanded to a double fan, i.e., two adjacencies of this vertex correspond to A and B, and the vertex is replaced by the n center vertices of the double fan. All other adjacencies of this vertex are connected to all of the center vertices. Note that this is possible, as the vertex has degree at least 2. Let x, y be the input strings, consisting of n bits each, of Alice and Bob for DISJOINTNESS, respectively.

Alice exposes to $\mathcal{A}$ all the vertices of G, except for the vertex B. Here, Alice exposes an edge between A and the i-th center vertex of the double fan if and only if the i-th bit of x is 1. Notice that Alice can expose all these vertices according to the VA model, as only the addition of the vertex B will require information of the input of Bob, y. If Alice has exposed all vertices of G except for B, then she passes the memory of $\mathcal{A}$ to Bob. Bob then exposes the vertex B, including an edge between B and the i-th center vertex of the double fan if and only if the i-th bit of y is 1. This completes the input to $\mathcal{A}$.

From the construction, observe that $|VC(G)| \leq K \leq |V(H)|$, as the vertex cover of G can always be bounded by taking all vertices originally in H, which covers the edges towards the n center vertices.

If the answer to DISJOINTNESS is NO, that is, there exists an index i such that $x_i = y_i = 1$, then in G the edges from A and B connect in the i-th center vertex, creating an induced copy of H in G, and so the graph is not H-free. Because $\ell = 0$, H-FREE DELETION [VC] must also be answered with NO.

If the answer to DISJOINTNESS is YES, that is, there is no index i such that $x_i = y_i = 1$, then there is no path from A to B directly through a center vertex in G. We will show that there is no induced occurrence of H in G. Name the minimal degree of vertices in H as d. Then the center vertices must have degree at most $d - 1$, as each center vertex cannot be adjacent to both A and B. But as d was the minimal degree in H, none of these center vertices can be used for an induced copy of H in G. But then G only has $|V(H)| - 1$ vertices remaining to form a copy of H, which is impossible. Hence, the answer to H-FREE DELETION $[K]$ is YES.

Now, from Proposition 4.29 it follows that any algorithm for solving H-FREE DELETION $[K]$ on a graph G with $K \geq |V(H)|$ requires at least n/p bits when using p passes in the VA/EA models, even when $\ell = 0$. This can be generalized for every ℓ by adding ℓ disjoint copies of H to G, which also increases the vertex cover of G by a constant amount for each copy. □

The following theorem is a generalization of the above Theorem 4.35.

Theorem 4.36. *If H is a graph with a vertex of degree at least 2 for which every neighbour has an equal or larger degree, then any algorithm for solving H-*FREE DELETION [VC] *on a graph G with $K \geq |V(H)|$ requires at least n/p bits when using p passes in the VA/EA models, even when the solution size $\ell = 0$.*

Proof. Similar to the reductions given by Bishnu *et al.* [8], we give a reduction from DISJOINTNESS to H-FREE DELETION [VC] in the VA model when the solution size parameter $\ell = 0$. The idea is to build a graph G with bounded vertex cover size, and construct edges according to the input of DISJOINTNESS, such that G is H-free if and only if the output of DISJOINTNESS is YES.

Let $\mathcal{A}$ be a one pass streaming algorithm that solves H-FREE DELETION [VC] in the VA model, such that $|\text{VC}(G)| \leq K \leq |V(H)|$, and the space used is $o(n)$. Let G be a graph with $n + |V(H)| - 1$ vertices consisting of H where a vertex degree at least 2 for which every neighbour has an equal or larger degree in H is expanded to a double fan, i.e., two adjacencies of this vertex correspond to A and B, and the vertex is replaced by the n center vertices of the double fan. All other adjacencies of this vertex are connected to all of the center vertices. Note that this is possible, as the vertex has degree at least 2. Let x, y be the input strings, consisting of n bits each, of Alice and Bob for DISJOINTNESS, respectively.

Alice exposes to $\mathcal{A}$ all the vertices of G, except for the vertex B. Here, Alice exposes an edge between A and the i-th center vertex of the double fan if and only if the i-th bit of x is 1.

Notice that Alice can expose all these vertices according to the VA model, as only the addition of the vertex B will require information of the input of Bob, y. If Alice has exposed all vertices of G except for B, then she passes the memory of $\mathcal{A}$ to Bob. Bob then exposes the vertex B, including an edge between B and the i-th center vertex of the double fan if and only if the i-th bit of y is 1. This completes the input to $\mathcal{A}$.

From the construction, observe that $|VC(G)| \leq K \leq |V(H)|$, as the vertex cover of G can always be bounded by taking all vertices originally in H, which covers the edges towards the n center vertices.

If the answer to Disjointness is NO, that is, there exists an index i such that $x_i = y_i = 1$, then in G the edges from A and B connect in the i-th center vertex, creating an induced copy of H in G, and so the graph is not H-free. Because $\ell = 0$, H-free Deletion [VC] must also be answered with NO.

If the answer to Disjointness is YES, that is, there is no index i such that $x_i = y_i = 1$, then there is no path from A to B directly through a center vertex in G. We will show that there is no induced occurrence of H in G. Let us call the vertex that was expanded into the center vertices v, and say it has degree $d \geq 2$. Then all the neighbours of this vertex must also have degree at least d in H. However, the center vertices in G have degree at most $d-1$, as no center vertex can be adjacent to both A and B. Hence, no center vertex can be used for v or any of its neighbours in an induced copy of H in G. Consider in H all vertices of degree at least d where the neighbours also have degree at least d, and say there are c many of these vertices. In any induced copy of H in G, these vertices must still have this relation of degrees. However, as none of the center vertices has degree at least d, G contains at most $c-1$ such vertices, which means an induced copy of H cannot occur in G. Hence, the answer to H-free Deletion [VC] is YES.

Now, from Proposition 4.29 it follows that any algorithm for solving H-free Deletion [VC] on a graph G with $K \geq |V(H)|$ requires at least n/p bits when using p passes in the VA/EA models, even when $\ell = 0$. This can be generalized for every ℓ by adding ℓ disjoint copies of H to G, which also increases the vertex cover of G by a constant amount for each copy. □

In Theorems 4.35 and 4.36 we only demand that the vertex cover size is at least $|V(H)|$, the number of vertices in H. One can wonder if this bound can be tightened, as in Theorem 4.34, where we only demand that the vertex cover size is at least $|\mathrm{VC}(H)| + 1$. The problem in Theorems 4.35 and 4.36, is that we might split a vertex of high degree. To get a valid vertex cover without it having linear size in n, the only option is to include at least all adjacencies of the center vertices. This makes it that the vertex cover can get a size up to $|V(H)|$, and so this is the only safe demand.

Let us shortly discuss why there is difficulty in obtaining more general lower bounds for H-free Deletion [VC].

It is important to notice that this reduction is not always correct. That is, there are instances for which naively picking some vertex to split into the double fan will result in an instance where a copy of H might occur while the answer to Disjointness is YES. An example of such an instance is given in Figure 4. The same problems more trivially occur for instances where H is a star, be it a P_3 or some star with more edges. The difference in star instances compared to instances such as those in Figure 4, is that star instances cannot work at all, while there is a choice for splitting a vertex that does lead to a correct reduction in the instance of Figure 4.

Let us give some insight in more possibilities for incorrect reductions. Another issue occurs when splitting a vertex of degree at least 4. This is because the adjacencies other than A and B will be made adjacent to all center vertices in the double fan. But then we can find new structures in this part of the graph. Namely, by traversing from a center vertex to one of these adjacencies, to another center vertex, to another of these adjacencies and then back to the first

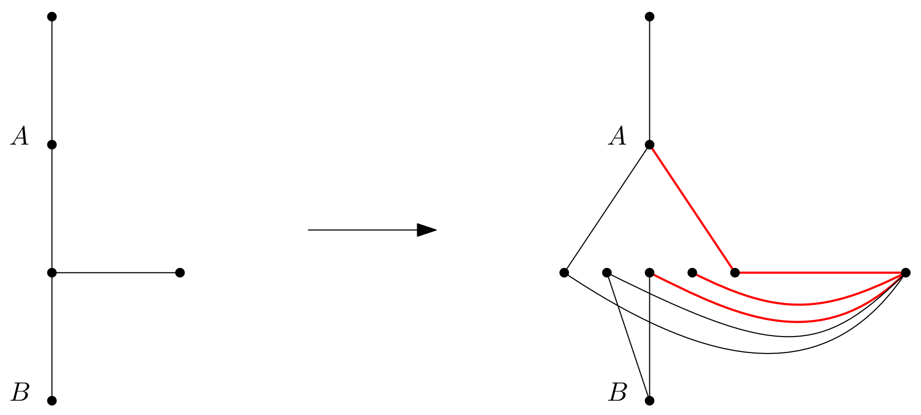

Figure 4: An illustration that the double fan technique can lead to an incorrect reduction. The left graph is H, where we split the degree 3 vertex, leading to the right picture. In the right picture, although Disjointness is YES, there is still an occurrence of H as indicated by the red edges. This counterexample is extendable by extending the path leading upwards from A.

center vertex, we can find a cycle. The size of such cycles are extendible depending on the degree of the vertex split, but can be both odd and even length if the adjacencies other than A and B might be adjacent to each other.

Although finding instances where these different occurrences directly make for an incorrect reduction is difficult, this does pose a problem for proving the correctness of general reductions. Arguments such as 'then there is no cycle in the graph' can be blatantly incorrect, falsifying an entire reduction. We are left in a dubious situation, as at this point it is unsure whether or not more general instances of H can be proven a lower bound, or whether there are very general instances where the reduction is simply incorrect. Further development might require a completely different lower bound technique or structure such that these problems do not occur.

It is also interesting to note that combining the AL model and a bounded vertex cover in a lower bound proof is problematic. In the instances we construct, we want to keep the vertex cover bounded, and so we cannot have the center vertices in the fan be center edges (or some larger structure), because we would get a vertex cover size of at least n. However, if these center vertices are merely vertices, then revealing their edges to an AL streaming model is something neither Alice nor Bob can completely do, because it requires information of the other party. Hence, it is difficult to combine these two factors in a lower bound proof.

Let us go over a few examples of problems that falling under Theorems 4.34, 4.35, and 4.36.

As any cycle as H has no degree-1 vertex, but also a degree-2 vertex, we can apply any of the three theorems to find that any algorithm in the VA model for H-free Deletion [VC] requires n/p bits when using p passes, when H is a cycle. As Odd Cycle Transversal [VC], Even Cycle Transversal [VC], and Feedback Vertex Set [VC] are generalizations of H-free Deletion [VC] where H is a cycle, these problems also require algorithms that solve them in the VA model to use n/p bits when using p passes, even with constant vertex cover size.

The problem of Cograph Deletion [VC] is equivalent to P_4-free Deletion [VC], and a P_4 has a degree-2 vertex and at least 3 edges. Hence, we can apply Theorem 4.34 to find that any algorithm in the VA model for Cograph Deletion [VC] requires at least n/p bits when using p passes, even with a constant vertex cover size.

5 Π-free Edge Editing

In this section we will consider a problem closely related to the previous Π-free Deletion, Π-free Edge Editing. Instead of vertex deletions, we allow edge additions and deletions in this problem variant. Once again, we will work with parameterization by vertex cover, as this has provided us with good results for Π-free Deletion already. However, because edge addition and vertex cover do not exactly go well together (adding an edge might make the vertex cover not a vertex cover anymore), we will require a second parameter, the solution size ℓ. In contrast to the previous section, $\ell \leq |VC|$ does not give us a trivial solution, and so we do not assume any relation between the size of the vertex cover and ℓ. All in all, these two parameters enable us to find new algorithms for Π-free Edge Editing in the streaming setting.

5.1 Problem Definition and Context

To formally define the problem, we first introduce an operation, the symmetric difference: Given two (edge) sets A and B, the symmetric difference $A \triangle B$ is the set of elements that are in either, but not both, sets. More formally, $A \triangle B = (A \setminus B) \cup (B \setminus A)$. This can also be interpreted as adding all the elements in B to A if they are not present yet, and removing from A those that are in B.

Let us give the formal definition of Π-free Edge Editing [VC, ℓ].

Π-Free Edge Editing [VC, ℓ]
Input: A graph $G = (V, E)$ with a vertex cover X, and an integer ℓ.
Parameter: The size $K := |X|$ of the vertex cover, the size of the solution ℓ.
Question: Is there a set $S = S_- \cup S_+ \subseteq (V \times V)$ of size at most ℓ such that $G' = (V, E \triangle S)$ does not contain a graph in Π as an induced subgraph?

Problems in the form of Π-free Edge Editing have extensive applications. One of the most well-known forms of this problem is P_3-free Editing, also known as Cluster Edge Editing. This problem has applications in biology or in general with missing data. Cluster Edge Editing asks us to add or remove edges of a graph to make it into a disjoint union of cliques, and any graph is a disjoint union of cliques if and only if it does not contain an induced P_3. The idea of the applications is that relational data usually forms cliques (that is, the relations are transitive), and thus we can 'fix' missing or corrupted data by solving the Cluster Edge Editing problem. An overview of the Cluster Edge Editing problem can be found in [10]. An overview of the general edge editing (modification) problem can be found in [22].

5.2 Cluster Editing

To get started with streaming algorithms for Π-free Edge Editing, we focus on one specific case: Cluster Edge Editing. Next to Cluster Edge Editing being a more specific version, it is also a well-used version of the problem, and so streaming algorithms for this problem are valuable results themselves. As mentioned before, Cluster Edge Editing [VC, ℓ] is equivalent to P_3-free Edge Editing [VC, ℓ], as a graph does not contain an induced P_3 if and only if it is a disjoint union of cliques.

There is a folklore branching algorithm for Cluster Edge Editing [ℓ], which finds any P_3 in the graph, and then branches on the three edits that remove this P_3 from the graph. This basic algorithm has a running time of 3^ℓ times some polynomial in n, and is FPT. The vertex cover parameter can come in useful here, because any P_3 must have at least one vertex in the vertex cover. Therefore, we present an adaptation of this simple branching algorithm to the streaming setting, using both vertex cover and solution size as parameters.

To be able to branch on a P_3 we first need to be able to find a P_3 in the graph, for which we give the following lemma.

Lemma 5.1. *In a graph $G = (V, E)$ given as a stream in the AL model with vertex cover X and edge modification set S, where $|X| = K$ and $|S| \leq \ell$, we can find an induced P_3 in $(V, E \triangle S)$ (if it exists) in $\mathcal{O}(K^2)$ passes using $\mathcal{O}((K + \ell) \log n)$ bits of memory.*

Proof. We assume that the vertex cover X and solutions set S are saved in memory, using $\mathcal{O}((K + \ell) \log n)$ bits of memory.

We attempt to find a P_3 in two separate phases. The first phase enumerates all of the K^2 edges/non-edges in the vertex cover. This can be be done with a dictionary ordering on all pairs of vertices in the vertex cover using $\mathcal{O}(\log n)$ bits of memory (See Definition 4.14, combined with the vertex cover in memory). Now consider one such an edge/non-edge in the enumeration, and let us fix the vertices that span this edge/non-edge as v and w. Any other vertex u is either adjacent to both, just one, or none of v and w. For a vertex u, we can identify which of these cases is present in the graph by the local information that is present for the edges of u in the stream, combined with the edits S makes. If $vw \in E \triangle S$, then u forms a P_3 if there is just one adjacency with v or w, and if $vw \notin E \triangle S$, then u forms a P_3 if it is adjacent to both v and w. Therefore, for a fixed pair v, w, we can identify if there is a P_3 containing v and w using a single pass and $\mathcal{O}(1)$ bits of memory. As we do this for every pair $v, w \in X$, we can conclude we can identify any P_3 with at least two vertices in the vertex cover using $\mathcal{O}(K^2)$ passes and $\mathcal{O}(\log n)$ bits of memory.

The second phase enumerates the other possibilities for a P_3 to appear. If we ignore S, then a P_3 cannot appear in the graph without any vertex in the vertex cover, as that would mean the vertex cover is not a vertex cover. Therefore, the only remaining possibility is for a P_3 to have one vertex in the vertex cover, which is adjacent to two vertices outside the vertex cover (which are by definition non-adjacent). So, to find a P_3 in phase two we only have to enumerate all vertices in the vertex cover and check whether it has two adjacencies outside the vertex cover (taking into account the edits made by S). This can be done entirely in one pass, as for each $v \in X$ we get all its adjacencies in the stream.

Now the only remaining issue is that the edits made by S might make the vertex cover invalid, and so can form a P_3 in ways we did not consider so far. To find such a P_3, we only have to enumerate all the edge additions in S, and for each one we need to check if it forms a P_3 (taking into account the other edits made by S). This check is similar to that of an edge within the vertex cover in phase one. Actually, we can find such an occurrence in a single pass, as for each vertex in the stream we consider its adjacencies towards the edge additions in S. As S is already in memory, we can use $\mathcal{O}(\ell)$ bits to save, for each edge uv added by S, if the current vertex in the stream is adjacent to one of the vertices u or v. If we see that the current vertex is adjacent to both or neither, there is no P_3. However, if at the end of the current vertex in the stream we have saved that there is a single adjacency towards one of these edge additions, this forms a P_3. So, we can find a P_3 occurring because of an edge addition made by S in $\mathcal{O}(1)$ passes using $\mathcal{O}(\ell)$ bits (assuming S is already saved in memory).

It should be clear that we can now find any occurrence of a P_3 in the graph with these enumerations. Phase two takes $\mathcal{O}(1)$ passes and $\mathcal{O}(\ell + \log n)$ bits of memory (if X and S are already in memory).

The above two phases enumerate all possibilities for a P_3 to appear in the graph, and so, find a P_3 if it exists. Phase one and two together use $\mathcal{O}(K^2)$ passes and require $\mathcal{O}((K + \ell) \log n)$ bits of memory. The lemma thus follows. □

We can now use Lemma 5.1 to get an algorithm for Cluster Edge Editing [VC, ℓ].

Theorem 5.2. *Given a graph G as a stream in the AL model with vertex cover X, we can solve* Cluster Edge Editing [VC, ℓ] *using $\mathcal{O}(3^\ell \cdot K^2)$ passes and $\mathcal{O}((K+\ell)\log n)$ bits of memory, where $|X| = K$ and ℓ is the solution size.*

Proof. The theorem essentially follows from repeatedly using Lemma 5.1 and branching on the possible edits. Let us elaborate on this process.

We start with $S = \emptyset$ and the vertex cover X in memory. Use Lemma 5.1 to find a P_3 in the graph. We now have three options, we can either remove one of the two edges of this P_3 and add it to S, or add an edge in place of the non-edge. For a side-note, if any of these edges already appear in S, than the edit will reverse a previous change, which is counterproductive. Therefore, ignore a branching option if it already occurs in S. For each of the successful branching options, continue this process by using Lemma 5.1 again. If at any point in a branch, Lemma 5.1 finds a P_3 but $|S| = \ell$, we stop in this branch. If at any point, Lemma 5.1 does not return a P_3 (because there is none), and $|S| \leq \ell$, we return S as a solution. If no branch finds a solution, we return NO.

Let us elaborate on the correctness of the above procedure. Notice that we always meet the preconditions of Lemma 5.1. Therefore, Lemma 5.1 correctly tells us whether or not there is a P_3 in the graph. If it does find a P_3, then this P_3 must be removed in a correct solution, and the algorithm branches on the only three ways to remove this P_3. Ignoring editing an edge if it already occurs in S is a correct operation, as reversing a previous edit is only counterproductive. If this reverse edit could lead to an optimal solution, then there is another branch at the place that made this original edit that leads to that optimal solution. Therefore, the algorithm works correctly.

The number of passes of the algorithm is dependent on the complexity of Lemma 5.1 together with the size of the search tree. Lemma 5.1 uses at most $\mathcal{O}(K^2)$ passes each time it is called. As we branch on three options every time, and we branch at most ℓ times, the total number of passes made by the algorithm is bounded by $\mathcal{O}(3^\ell \cdot K^2)$.

For memory use, the algorithm only needs to save the sets X and S, which take $\mathcal{O}((K + \ell)\log n)$ bits of space, and the use of Lemma 5.1 does not exceed this complexity. Also, we branch on a constant number of options in each step, which means we asymptotically need $\mathcal{O}(\ell \log n)$ memory to remember branching sets.

We can conclude that the theorem follows. □

Now that we have seen a basic branching algorithm for Cluster Edge Editing [VC, ℓ] in the streaming setting, we can wonder whether we can do better. We can try to, for example, look at existing branching algorithms for Cluster Edge Editing and see if their branching strategies can be applied in the streaming setting. This turns out to be the case, we will look to improve our branching strategy based on that of Gramm *et al.* [34].

In Section 4.1 Gramm *et al.* give three branching rules (B1), (B2), and (B3) which correspond to the branching rules we use in Theorem 5.2, namely, if we have some P_3, we branch on deleting either one of the edges ((B1) and (B2)), or add the missing edge ((B3)). In Section 4.2 Gramm *et al.* give an extension on these branching rules to improve on the size of the search tree produced by the algorithm. Next we will show that these refined branching rules can be adapted to the streaming setting as well.

For completeness, we first give the cases for the refined branching rules as given by Gramm *et al.* [34].

Given a P_3 consisting of vertices u, v, w where $uv \in E$ and $uw \in E$ but $vw \notin E$, we have three situations:

(C1) Vertices v and w do not share a common neighbour, so $\nexists x \in V, x \neq u : vx \in E$ and $wx \in E$.

(C2) Vertices v and w have a common neighbour $x \neq u$, and $ux \in E$.

(C3) Vertices v and w have a common neighbour $x \neq u$, and $ux \notin E$.

The first thing to notice here, is that we can identify which case we are in using a single pass of the AL stream. Furthermore, Gramm *et al.* provide deterministic branching for each of these cases, and we can simply follow up on that process. For completeness, we summarize the branching rules for each case next. Remember, we have found a P_3 with vertices u, v, w in our graph such that $uv \in E$ and $uw \in E$ but $vw \notin E$. We also identified which of cases (C1), (C2), (C3) is present, using a pass over the stream. We get the following branching rules:

(C1) In this case, we get a regular branch, except that the edge addition branch can be left out. This is proven in [34, Lemma 5]. So we only branch on deleting uv or uw.

(C2) In this case, we do the three normal branches, but we can expand the branching in two of the cases further. This leads to the following branches:

- Add the edge vw.
- Delete the edge uv, and also delete vx.
- Delete the edge uv, and also delete ux and wx.
- Delete the edge uw, and also delete wx.
- Delete the edge uw, and also delete vx and ux.

(C3) In this case, we do the three normal branches, but we can expand the branching in two of the cases further. This leads to the following branches:

- Delete the edge uv.
- Delete the edge uw, and also delete vx.
- Delete the edge uw, and also delete wx but add ux.
- Add the edge vw, and also delete wx and vx.
- Add the edge vw, and also add ux.

Clearly, we can use these branching rules instead of those in Theorem 5.2 and only increase the number of passes by one in each branch. Also, the number of branches we create in each step is still bounded by a constant, just as the edits each branch makes, which means we do not increase the asymptotic memory use.

Before we come to the complete result, however, it is good to notice that we do one thing different from Gramm *et al.*. We do not mark edges as 'permanent' or 'forbidden'. We have to ask ourselves if the algorithm then still works correctly, and if it still results in the same bounded size search tree. The purpose of marking edges as permanent or forbidden, is that in a later branch we do not undo the work we did previously. One can imagine removing a P_3 with some edit, which can create another P_3, which we then edit such that the first P_3 is back in the graph in some branch. This can be a problem in three possible ways: it might make the algorithm loop indefinitely, it might increase the search tree size, and it might make the algorithm incorrect. The indefinite looping is quickly resolved: if we still decrease the parameter effectively with each edit, we are still bounded by the parameter in the number of edits we can make. In the algorithm, this means that the solution set S may contain duplicates, and we must allow this. Because the parameter is still correct, the worst case search tree cannot increase in size. This simply because in the worst case, this scenario does not even have to occur, as we have enough P_3 occurrences that do not overlap anyway, which all require branching. Therefore, the worst case does not change, and so the complexity does not change either. The last concern was correctness. To show this, consider the following. If we do all the marking into permanent and forbidden vertices, then some branching steps do less branching, as some branches undo previously done work. In this case, the algorithm works correctly. In our case, we do expand these branching steps, which means our algorithm only considers more options, including all those considered by the correct-marking algorithm. Therefore, if a solution exists, our algorithm also considers it, as some branching order corresponds to the edits made in the solution.

With the above motivation, we can safely adjust Theorem 5.2 with the extended branching rules, and using [34, Theorem 2], we come to the following theorem:

Theorem 5.3. *Given a graph G as a stream in the AL model with vertex cover X, we can solve* Cluster Edge Editing [VC, ℓ] *using $\mathcal{O}(2.27^\ell \cdot K^2)$ passes and $\mathcal{O}((K+\ell)\log n)$ bits of memory, where $|X| = K$ and ℓ is the solution size.*

5.2.1 Cluster Editing Kernels

To obtain more results, we can look into adapting a kernel to the streaming setting. In exploring kernels, this adaptation proves to be far from trivial, as many kernels use graph properties or structures that are not easily computed in the streaming model with sub-linear memory. However, by making use of the vertex cover parameter we can remain memory efficient in adapting such a kernel. The kernel we will adapt is that of Jiong Guo [35], and in particular, the 6ℓ kernel provided. In short, the 6ℓ kernel first computes the Critical Clique Graph of the input, over which it has some rules depending on the size of neighbourhoods to reduce the instance. As mentioned, this adaptation is not trivial, and we require multiple careful analyses to accomplish this result. Our first step is to be able to compute the Critical Clique Graph of our input.

Let us first give the definition of a critical clique and a critical clique graph, just as they were given in [35].

Definition 5.4. A *critical clique* of a graph G is a clique C where the vertices of C all have the same set of neighbours in $V \setminus C$, and C is maximal under this property. A critical clique is also known as a set of *true twins*.

Definition 5.5. Given a graph $G = (V, E)$, let $\mathcal{K}$ be the collection of its critical cliques. Then the *critical clique graph* $\mathcal{C}$ is a graph $(\mathcal{K}, E_\mathcal{C})$ with $C_iC_j \in E_\mathcal{C} \iff \forall u \in C_i, v \in C_j : uv \in E$.[2]

On a graph in memory we can compute the critical clique graph by partitioning the vertices on their closed neighbourhood. However, in a streaming setting this is much more difficult. First we have the problem of memory: the critical clique graph can have $\mathcal{O}(n)$ nodes and $\mathcal{O}(m)$ edges, and furthermore, we cannot simply partition on a closed neighbourhood, as this might consist of $\mathcal{O}(n)$ vertices. Thankfully, if we use the vertex cover we can make a useful observation: any two vertices outside the vertex cover cannot be in the same critical clique, as there is no edge between them. This means we can only have $\mathcal{O}(K)$ critical cliques with a size bigger than one, as it must consist of a part of the vertex cover with at most one vertex outside the vertex cover. This leads to the following idea: we can save only the critical cliques contained in the vertex cover and those with size bigger than one using $\mathcal{O}(K \log n)$ memory. Any other vertex in the stream that is not saved in such a critical clique must therefore be a critical clique on its own.

[2]In more modern terminology, a critical clique is a module in the graph. The critical clique graph can then be seen as a quotient graph of this modular decomposition. As this is very general terminology that is not required for the contents of this thesis, we will not go into the definitions and details of modules and modular decompositions.

Hence, we have enough information saved so that we can inspect the critical clue graph $\mathcal{C}$ using a pass over the stream. Let us formalize this in a lemma.

Lemma 5.6. *Given a graph G as an AL stream together with a vertex cover X of G, $|X| = K$, we can compute and save (the vertices of) all critical cliques of G that are either contained in the vertex cover or of size bigger than one using $\mathcal{O}(1)$ passes and $\mathcal{O}(K \log n)$ bits of memory.*

Proof. Let $G = (V, E)$ be a graph with vertex cover X such that $|X| = K$. We will first partition the vertex cover into the critical cliques contained in it, and then consider options for vertices outside the vertex cover that might belong to one of these critical cliques, making it maximal.

We first view the vertex cover X as the entire graph, and partition it into its critical cliques. This can be done by a process described in [38, Theorem 3.3], where we iteratively refine the current partitions with a vertex v by splitting each partition P on $P \cap N[v]$ and $P \setminus N[v]$. Notice that we need only one pass to do this for the vertex cover X, as for each $v \in X$ finding $N[v] \cap X$ requires local information to v in the AL stream and only $\mathcal{O}(K \log n)$ bits of memory. The set of partitions we save can have no more than K elements making up K vertices in total, which means we only need $\mathcal{O}(K \log n)$ bits of memory. With this one pass we have found the partitioning of X into its critical cliques, but this ignores all adjacencies towards vertices not in X.

So, what remains is two problems. Firstly, we might have to partition critical cliques in X further because some vertices do not share the same adjacencies towards $V \setminus X$. Secondly, some critical cliques might include a vertex in $V \setminus X$, which we will have to find as well.

The first problem can be solved in another pass. In this pass, for each vertex $v \in (V \setminus X)$, we split each partition on $N(v)$, that is, a set P is partitioned into $P \cap N(v)$ and $P \setminus N(v)$. This process splits up all critical cliques which do not share the same adjacencies towards $V \setminus X$. Notice that $|N(v)| \leq K$ for $v \in (V \setminus X)$, and so, this can be done using $\mathcal{O}(K \log n)$ bits of memory.

At this point, each critical clique can be missing at most one vertex now, which can be a vertex in $V \setminus X$. This can only be one vertex, as no critical clique can contain two vertices from $V \setminus X$, as those vertices form an independent set. We use two passes to find such vertices. In the first pass, for each vertex of a critical clique, if it has a single adjacency outside of X, we save this adjacent vertex as a potential candidate to belong in this critical clique. This first pass nets us $\mathcal{O}(K)$ candidate additions to critical cliques, at most one per critical clique. We save these candidates with their respective critical cliques in (critical clique, candidate)-pairs, which together use $\mathcal{O}(K \log n)$ bits of memory. In the second pass, whenever we see a vertex v in the stream, consider its adjacencies towards the critical cliques and the candidate additions. If this vertex is adjacent to both or neither a (critical clique, candidate)-pair, the candidate can still belong to the critical clique. If it is not, that is, v is adjacent to one of the critical clique or the candidate, we throw away this candidate, as it cannot belong to the critical clique. We can do this for all of the $\mathcal{O}(K)$ (critical clique, candidate)-pairs by locally saving $\mathcal{O}(K)$ bits of information when considering the vertex v, by saving boolean bits for the presence of adjacencies towards (the elements of) these pairs. When we go to the next vertex w in the stream, we forget the locally saved information, and do the same process again for w. After the pass, we have thrown away all candidates that do not share some adjacency with the corresponding critical clique, and hence, all remaining candidates belong to the respective critical clique. Therefore, save these candidates as permanent elements of the critical cliques.

We claim that we now have (the vertices of) all critical cliques contained in the vertex cover, or of size bigger than one, saved. Note that the above process takes $\mathcal{O}(1 + 1 + 2) = \mathcal{O}(1)$ passes over the stream and does not exceed $\mathcal{O}(K \log n)$ bits of memory usage. Let us argue why we have all such critical cliques saved. As we partitioned the vertex cover correctly by [38, Theorem 3.3], we have all critical cliques contained in the vertex cover. Because no two vertices outside the vertex cover can belong to the same critical clique, all critical cliques of size bigger than one can only consist of some subset of X (possibly) together with a vertex outside X. But then this vertex outside X must be the only adjacency of the subset of X of this critical clique, and so it is considered in the process. Therefore we have all such critical cliques in memory. $\square$

As argued before, executing Lemma 5.6 enables us to use a pass over the stream to inspect the critical clique graph, as its complete structure can be derived from the information in the stream together with the saved critical cliques.

The 6ℓ kernel of Guo [35] follows from the critical clique graph by applying the following two rules:

Rule 1 Remove any isolated critical clique from $\mathcal{C}$.

Rule 2 If there is a critical clique $C \in \mathcal{C}$ such that $|V(C)| > |\bigcup_{C' \in N_{\mathcal{C}}(C)} V(C')| + |\bigcup_{C' \in N^2_{\mathcal{C}}(C)} V(C')|$, where $N_{\mathcal{C}}(C)$ and $N^2_{\mathcal{C}}(C)$ denote the first and second neighbourhood of C, then we edit $G[\bigcup_{C' \in N_{\mathcal{C}}(C)} V(C')]$ into a complete graph and delete all edge between $\bigcup_{C' \in N_{\mathcal{C}}(C)} V(C')$ and $\bigcup_{C' \in N^2_{\mathcal{C}}(C)} V(C')$ and decrease ℓ by the corresponding amount. This means we have made $C \cup \bigcup_{C' \in N_{\mathcal{C}}(C)} V(C')$ into a complete graph disconnected from all other vertices, and so we can remove it according to Rule 1.

Adapting these kernelization rules to the streaming setting provides us with a couple of problems. We do not simply have the graph in memory with the option to edit it until we have the kernel. Rather, we want to compute what our kernel is after applying the rules, and output it then. Secondly, deciding the size of the first and second neighbourhood, and the necessary edits, is also far from trivial, and must be approached carefully if we wish to execute this using little memory. Let us first go into detail how we compute and execute Rule 2 on basis of the information saved by Lemma 5.6.

Lemma 5.7. *Given a graph G as an AL stream together with a vertex cover X of G, $|X| = K$, and the critical cliques (partially) contained in X, and an edit set S with $|S| \leq \ell$, we can decide whether Rule 2 applies to a given critical clique C, and execute it, using $\mathcal{O}(1)$ passes and $\mathcal{O}((K + \ell)\log n)$ bits of memory.*

Proof. In this lemma, whenever we consider some edges/non-edges, we assume we have taken into account the edits made by S. Looking at any edge/non-edge, this can always be done using $\mathcal{O}(|S|)$ time, which is insignificant as we allow unbounded computation time.

In the following analysis, we work on the critical clique graph $\mathcal{C}$. Remember that we have all critical cliques that are (partially) contained in X saved. Therefore, we can always check if we have 'done' a critical clique already, by checking if the current vertex in the stream belongs to a critical clique with a vertex we have seen earlier in the stream (and so always mark any of these saved critical cliques as 'done' if any vertex of it is handled in the stream). As other critical cliques only consist of one vertex, Rule 2 will never apply to them. This is because such a critical clique is not isolated, as otherwise it would be removed by Rule 1. Hence, it has a neighbour in the vertex cover. But then this neighbour has at least one vertex, which means Rule 2 cannot apply. By this motivation, it should be clear that we can view a pass over the stream to be equivalent to a pass over the critical clique graph.

Deciding on the exact first and second neighbourhood using little memory is hard. However, counting their size is easier. To do this, we start with two passes over the stream. In the first pass, we mark all neighbours of the given critical clique C that are within X as being first neighbours of C. In the second pass, we mark all second neighbours of C as being second neighbours, by marking all unmarked neighbours of the marked neighbours of C. We can now count the number of first and second neighbours of C (of course, ignoring C itself). We use a pass over the stream and do the following. If the current node is in X and it is marked, we increment the corresponding counter. If the current node u is outside of X, we look at its adjacencies, and count the number of unmarked neighbours (ignoring C itself). If C is a neighbour of u, then these unmarked neighbours are second neighbours and must be counted. If u is not a neighbour of C, but is a neighbour of a first neighbour, then it is a second neighbour and it must be counted. After this pass, we have counted all first and second neighbours, as any first-second neighbour pair must have at least one of the two in X.

Notice that we now have enough information to decide whether to apply Rule 2, using $\mathcal{O}(1)$ passes and $\mathcal{O}(K \log n)$ bits of memory. Let us now assume that the conditions are met, and we want to do the editing as suggested by Rule 2.

We have to add edges such that all first neighbours form a clique together with $\mathcal{C}$. Note that we already know how many first neighbours are in X and how many first neighbours we have total. Therefore, we know how many first neighbours of C lie outside of X. Notice that these have no edges between them, so if there are i of such nodes, we require at least $(i^2 - i)/2 = \mathcal{O}(i^2)$ edge additions. But, we are allowed to make at most ℓ edits. We can conclude that $i^2 = \mathcal{O}(\ell)$ and otherwise we return NO. Now we can simply use a pass to locally save all these first neighbours, and then use a pass to find all the missing edges between them. Add these edges to S, unless $|S| > \ell$, then return NO. Note that this process takes $\mathcal{O}(1)$ passes and uses $\mathcal{O}((K + \ell) \log n)$ bits of memory.

What remains is to remove all the edges between the first and second neighbours of C. However, we have saved all the first neighbours of C in memory, so this process is not difficult. Use a pass over the stream, and at every node, if it is a second neighbour (so it is adjacent to a first neighbour but not C), add all its adjacencies to first neighbours to S as removals. If $|S| > \ell$, then return NO.

It should be clear from the process that we correctly decide whether to apply Rule 2, and correctly apply Rule 2 if needed, using in total $\mathcal{O}(1)$ passes and $\mathcal{O}((K + \ell) \log n)$ bits of memory. □

What remains to be able to completely adapt the kernel to the streaming setting, is two questions. How many times can we apply Rule 2 and Rule 1, and how can we make sure we actually get the kernel. For the first matter, notice that we can ignore Rule 1 until we actually want to output the kernel, as it makes no edits, and Rule 2 can only be applied for a critical clique C with $|C| > 1$, as otherwise it only applies when a critical clique is isolated. Therefore, we only have to check to apply Rule 2 on the saved critical cliques, which are $\mathcal{O}(K)$ many. Because the edits made can make the rule apply to a previously checked critical clique, we have to execute Lemma 5.7 at most $\mathcal{O}(K^2)$ or $\mathcal{O}(K\ell)$ times. This is because there are only $\mathcal{O}(K)$ many cliques to apply the rule to, and each application makes at least one edit, and we never apply the rule to the same clique twice.

The second question is how we can find the actual kernel after applying Rule 2 exhaustively. The graph now consists of a disjoint union of isolated cliques and a 6ℓ kernel. What we can do to solve this matter is simple: every time we apply Rule 2 to some critical cliques, we mark them as being isolated. We have remaining some subset of the saved critical cliques which are not isolated, and we can use a pass to decide, for every vertex outside X, whether it belongs to some isolated clique or is adjacent to one of these non-isolated critical cliques (respecting the solution set S).

The attentive reader might notice a possible issue: the edits made by S change the vertex cover and the critical clique graph, while we seem to ignore this completely. That is, edges added by S may be edges not covered by the vertex cover, and the edges added and deleted by S increase the size of some critical cliques. However, we argue that this does not matter. First notice that we do respect the edits made by S in inspecting the stream in Theorem 5.7. Also notice that applying Rule 2 isolates a critical clique (which now might have increased in size). As we just mentioned that we will mark the saved critical cliques as being isolated, and an isolated critical clique will never receive any more edges, we can safely ignore these saved critical cliques while applying Rule 2. Even better, if we respect the edits made by S, the vertices in this (isolated) critical clique will never be reachable, even those vertices that were added later and are not saved as being part of this critical clique. The edges added by S that might make the vertex cover invalid are also only contained in isolated critical cliques, which means we will not need to look at these edges at any time. Hence, it is safe to work only work with the originally saved critical cliques, and only respecting the edge additions and deletions made by S when inspecting the stream.

This should be enough motivation to lead to the following theorem.

Theorem 5.8. *Given a graph G as an AL stream with a vertex cover X, $|X| = K$, we can find a 6ℓ kernel for* Cluster Edge Editing [VC, ℓ] *using $\mathcal{O}(\min(K^2, K\ell))$ passes and $\mathcal{O}((K + \ell)\log n)$ bits of memory.*

Proof. First use Lemma 5.6, which uses $\mathcal{O}(1)$ passes and $\mathcal{O}(K \log n)$ bits of memory. Then exhaustively apply Lemma 5.7 on these saved critical cliques to exhaustively apply Rule 2. If we successfully apply Rule 2, we mark the now-isolated nodes as being isolated (we mark only those already saved by Lemma 5.6). We can apply Lemma 5.7 at most $\mathcal{O}(K^2)$ or $\mathcal{O}(K\ell)$ times, depending on which is smaller, which is $\mathcal{O}(\min(K^2, K\ell))$ times. Now use a pass to find the kernel by outputting all vertices and edges which contained in, or adjacent to, some non-marked critical clique, ignoring the edges deleted by S. Clearly, this process takes $\mathcal{O}(\min(K^2, K\ell))$ passes of the stream and uses $\mathcal{O}((K + \ell)\log n)$ bits of memory. The correctness and size of the kernel follows from the correctness of Rule 1 and Rule 2, as proven by Guo [35]. □

It is also possible to adapt the 4ℓ kernel of Guo [35] to the streaming setting. However, it is questionable whether this is worth it over the 6ℓ kernel. This is because the rules for the 4ℓ kernel are more specific, meaning they might be applied much more often than those of the 6ℓ kernel. Because we cannot do ‘partial’ passes, a pass is a pass, this would lead to many more passes necessary to apply these rules. To be more specific, Rule 3 for the 4ℓ kernel in [35] asks for critical cliques K and K' such that their size together meets a demand similar to that of Rule 2. However, we can observe that the critical clique K must be in X, but not necessarily the critical clique K'. This would mean we possible have to apply this rule to many critical cliques K', or at least, check whether it applies. These checks require passes and so the number of passes will likely increase significantly. Therefore, we leave this adaptation as a remark and do not describe its details in full.

Let us move on to a different kernelization. It seems from the above kernelization that the vertex cover provides us with a lot of flexibility to work with memory and information. It might therefore be interesting to think about a simple kernelization strategy which makes use of the given vertex cover to derive structural information. One approach to accomplish this is to analyse the graph with a given size of both the vertex cover and the independent set. This simple idea leads to the following lemma.

Lemma 5.9. *Given an n-vertex graph G with vertex cover X, $|X| = K$, if there are no isolated vertices and $n = 2K + c$, then it takes at least c edge edits to make the graph into a disjoint union of cliques.*

Proof. We will now show that for any K, a graph with $n = 2K + c$ non-isolated vertices requires at least c edits. Let G be a graph meeting the preconditions, and let S be a minimal set of edits which makes G into a disjoint union of cliques. Let us denote $G' = (V(G), E(G) \triangle S)$, the resulting graph after applying the edits of S to G.

To prove the lemma, we consider the cases of a vertex $v \notin X$ with respect to the cliques in G'. It can be that v is alone in a clique in G', which requires at least one edit because G contains no isolated vertices. Another case is that v is in a clique with multiple vertices from outside X, which requires edge additions as these vertices originally form an independent set. The remaining scenario is that v is in a clique with only some subset of vertices from X. This last scenario might not require edits. However, we can bound how often this scenario occurs, as it requires at least one vertex from X and at most one vertex from outside X. Say there are $d \leq K$ of such cliques in G'. Now there are at most $K - d \geq 0$ vertices remaining in X, and at least $K + c - d \geq c$ vertices outside of X. This means there is still a difference of at least c vertices in the size of X and its complement (note that if some of these d cliques contain multiple vertices from X, this difference is only larger). Now, for each $v \notin X$, the clique in G' containing v is only of two cases: either v is a clique on its own, or v is in a clique with at least one other vertex from outside X. Every vertex in the first case requires at least one edit, as G contains no isolated vertices. Every clique in G' with $q \geq 2$ vertices from outside X requires at least $q - 1$ edits in S, because these vertices are an independent set. Because there are still at least c more vertices outside X than in it, S must contain at least c edits to make these cliques. Therefore, the lemma follows. □

It is good to note that the bound of $n = 2K + c$ vertices is tight. If we consider $c = 0$ there is indeed an instance requiring no edits. This instance is simply K disjoint edges, each of which has one vertex in the vertex cover. As this instance is a disjoint union of cliques, it requires no edits.

The following kernelization theorem follows directly.

Theorem 5.10. *Given a graph G in the VA/AL streaming model with a vertex cover X, $|X| = K$, we can get a $2K + \ell$ kernel for* Cluster Edge Editing [VC, ℓ] *using one pass and $\mathcal{O}((K + \ell) \log n)$ bits of memory (to save the kernel).*

Proof. Use a pass to save all non-isolated vertices. If at any point the number of vertices we save exceeds $2K + \ell$ we can conclude from Lemma 5.9 that there exists no solution. □

5.3 Π-free Editing with explicit Π

Let us move on to the general Π-free Editing. To give an algorithm for this problem, we look towards a very general result of Cai [13], that for finite Π graph modification problems are fixed-parameter tractable. The result that Cai provides is not immediately usable in the streaming setting, as the procedure used to find a minimum forbidden induced subgraph is memory expensive. Luckily, in previous sections we have already seen some tools and algorithms which enable us to build an algorithm inspired on that of Cai. To give this algorithm, we first give some general notation as used by Cai [13]. Notice that the Cai works on a more general Π-free Editing problem, where we also allow vertex deletions. To distinguish these versions, we will call Π-free Edge Editing the variant which does not allow vertex deletions, and Π-free Editing the variant which does.

If we assume that Π is a finite set of forbidden induced subgraphs, we can denote the maximum size of a graph in Π with ν. Because the result is for general modification problems, we separate the number of edits we can make into three variables, where i is the number of vertices we are allowed to delete, j the number of edges we can delete, and k the number of edges we can add. Note that for Cluster Edge Editing we essentially had that $\ell = j + k$ (for some j, k) and $i = 0$, while in Section 4 we had $\ell = i$ and $j = k = 0$.

In essence the algorithm of Cai [13] is very simple: find a minimum size induced subgraph from Π in the graph G, and branch on all possible edits. The issue in adapting this method to the streaming setting is finding a minimal induced subgraph. Luckily, in Section 4.3.2 we have seen a method perfectly suitable for finding induced an induced subgraph given a vertex cover, FindH (see Algorithm 6 and Algorithm 7). The only adjustment we have to make, is to make FindH return the entire instance of H it found instead of only the vertices outside the vertex cover.

From the above analysis and description, the following theorem follows.

Theorem 5.11. *If Π is a graph property such that the maximum size of any graph in Π is ν, and given a graph G as an AL stream with a vertex cover X, where $|X| = K$, we can solve* Π-free Editing [VC, i, j, k] *where we can make i vertex deletions, j edge deletions, and k edge additions, using $\mathcal{O}(|\Pi| \cdot \nu^{i+2j+2k+1}(K+k)^{\nu}\nu!)$ passes and $\mathcal{O}((K + (i+j+k)\nu^2)\log n + |\Pi|\nu^2)$ bits of memory.*

Proof. The idea of the algorithm is the following: take the smallest graph in Π, and call FindH at most $\nu + 1$ times to check whether there is an occurrence of this graph in G (once for every $x = 0 \dots \nu$, where x is the parameter which makes FindH look for an instance with x vertices outside of the vertex cover). If we find an occurrence, branch on all the different edits we can make to this occurrence, respecting i, j, k. In each branch, start this process again. If we do not find the smallest graph in Π, try the second smallest, etc.. The total number of branches then comes down to $\binom{\nu}{2}^{j+k}\nu^i = \mathcal{O}(\nu^{i+2j+2k})$. If at any time we add an edge that makes the vertex cover invalid, we add one of the two vertices to the vertex cover, such that the vertex cover remains valid with the edits made.

The correctness of the algorithm follows from the correctness of FindH as given in Lemma 4.16, combined with the branching being a simple 'try everything' approach.

The number of passes of this algorithm is dependent on the size of Π, the number of branches, and the number of passes FindH makes. This comes down to $\mathcal{O}(|\Pi| \cdot \nu^{i+2j+2k+1}(K+k)^{\nu}\nu!)$ passes, where the $(K + k)$ comes from the fact that we expand the vertex cover of size K with at most k vertices, and it is to the power of ν because of the complexity of FindH, where its parameter i is at most ν.

The memory use is a little more complicated. Clearly, we need to have Π saved, together with the vertex cover. To be able to branch, we need the entire occurrence of any forbidden induced graph we find to be in memory, and this simultaneously for every active instance, which can be $\mathcal{O}(i + j + k)$ many. We also need to keep track of the edits we made. The size of the rest of possible sets we need can be bounded by one of these sets, and so the total memory use is bounded by $\mathcal{O}((K + (i+j+k)\nu^2)\log n + |\Pi|\nu^2)$ bits. □

It is interesting to note that this gives us an alternative for Algorithm 4.3 and Algorithm 7 by setting $j = k = 0$ and $i = \ell$. Although the asymptotic performance of Theorem 5.11 might look better than that of Algorithm 7, it does not necessarily perform better. This is because in Algorithm 7 some work is done to attempt to lower the number of branches made, but is worst case still h, while Theorem 5.11 always branches on $\mathcal{O}(\nu + \binom{\nu}{2})$ options.

The attentive reader might notice that the problem definition has slightly shifted. We originally specified that a total of ℓ edits could be made, either edge additions or deletions. However, in Theorem 5.11 we explicitly have how many additions and how many deletions we may make. We could solve this by applying Theorem 5.11 for all j, k such that $j + k = \ell$, but this would slightly increase the complexity. A better alternative, is to edit the algorithm to work with ℓ, where instead of getting $\binom{\nu}{2}^{j+k}\nu^i = \mathcal{O}(\nu^{i+2j+2k})$ total branches we get $\binom{\nu}{2}^{\ell} = \mathcal{O}(\nu^{2\ell})$ branches. So instead of each edit having a specific parameter to decrease when made, both the edge edits now decrease ℓ, and the vertex deletion edit is illegal. This leads to the following theorem.

Theorem 5.12. *If* Π *is a graph property such that the maximum size of any graph in* Π *is* ν*, and given a graph* G *as an AL stream with a vertex cover* X*, where* $|X| = K$*, we can solve* Π-FREE EDGE EDITING [VC, ℓ] *where we can make a total of* ℓ *edge additions or deletions, using* $\mathcal{O}(|\Pi| \cdot \nu^{2\ell+1}(K+\ell)^{\nu}\nu!)$ *passes and* $\mathcal{O}((K+\ell\nu^2)\log n + |\Pi|\nu^2)$ *bits of memory.*

5.4 Π-free Editing without explicit Π

Just like with Π-FREE DELETION, we can asks ourselves whether we can achieve similar results without saving Π explicitly, which should (hopefully) decrease the memory use of the algorithms. As before, we want to work with an oracle algorithm, which on call, gives us some information about Π.

However, it seems we run into the same problems in using an oracle algorithm for Editing problems. The algorithm of Theorem 5.11 relies quite heavily on FINDH, and that function requires explicit knowledge of the graph it is searching. Interestingly, Cai [13] provides an algorithm for finding a minimal forbidden induced subgraph using some oracle algorithm that tells us whether a graph G is Π-free or not. We can be tempted to adapt this method to the streaming setting. However, the method requires a set of size $\mathcal{O}(n)$ to be in memory. This is far from ideal if we want to save memory by not explicitly saving Π.

It seems the options are very limited. To give at least some algorithm, we can consider a brute force approach. Say we work with oracle algorithm $\mathcal{A}_1$, that is, when $\mathcal{A}_1$ is called on a graph G, it returns whether or not $G \in \Pi$. As we know that every graph in Π has a maximum size ν, we can simply iterate over all $\binom{n}{\leq \nu} = \mathcal{O}(\nu n^{\nu})$ options and call the oracle algorithm on each option. Whenever the oracle returns that the graph is in fact a forbidden graph, we can remove it by branching on all possible edits. This brute force approach clearly leads to a correct solution, if it exists. If we denote $P_{\mathcal{A}_1}(n), M_{\mathcal{A}_1}(n)$ as respectively the number of passes and the memory use of the oracle algorithm $\mathcal{A}_1$ when called on a graph G of size n, it is not hard to see that this algorithm would use $\mathcal{O}(n^{\nu}\nu^{i+2j+2k+1}(1+P_{\mathcal{A}_1}(\nu)))$ passes. If we use a dictionary ordering on all of the $\binom{n}{\leq \nu}$ options, we can achieve a memory use of $\mathcal{O}((i+j+k)\nu^2 \log n + M_{\mathcal{A}_1}(\nu))$ bits. We multiply $(i+j+k)$ with $\nu^2 \log n$ here, because to be able to continue branching when returning out of recursion, we need to remember the edits we want to make. As the search tree has depth at most $(i+j+k)$, the memory use follows.

This leads to the following theorems.

Theorem 5.13. *If* Π *is a graph property such that the maximum size of graphs in* Π *is* ν*, and* $\mathcal{A}_1$ *is an oracle algorithm that, when given subgraph* H *on* h *vertices, tells us whether or not* H *is in* Π *using* $P_{\mathcal{A}_1}(h)$ *passes and* $M_{\mathcal{A}_1}(h)$ *bits of memory, we can solve* Π-FREE EDITING $[i,j,k]$ *where we can make i vertex deletions, j edge deletions, and k edge additions using* $\mathcal{O}(n^{\nu}\nu^{i+2j+2k+1}(1+P_{\mathcal{A}_1}(\nu)))$ *passes and* $\mathcal{O}((i+j+k)\nu^2 \log n + M_{\mathcal{A}_1}(\nu))$ *bits of memory.*

Theorem 5.13 implies the following result for the Π-FREE DELETION [ℓ] problem.

Theorem 5.14. *If* Π *is a graph property such that the maximum size of graphs in* Π *is* ν*, and* $\mathcal{A}_1$ *is an oracle algorithm that, when given subgraph* H *on* h *vertices, tells us whether or not* H *is in* Π *using* $P_{\mathcal{A}_1}(h)$ *passes and* $M_{\mathcal{A}_1}(h)$ *bits of memory, we can solve* Π-FREE DELETION [ℓ]*, where we can make* ℓ *vertex deletions, using* $\mathcal{O}(n^{\nu}\nu^{\ell+1}(1+P_{\mathcal{A}_1}(\nu)))$ *passes and* $\mathcal{O}(\ell\nu^2 \log n + M_{\mathcal{A}_1}(\nu))$ *bits of memory.*

6 Vertex Cover

The Vertex Cover problem is a well-known problem with much attention in the literature [24, 31, 55]. In previous sections, we used vertex cover as a parameter. This motivates us to explore how well we can do in finding a vertex cover in the streaming setting.

6.1 Problem Definition and Context

Let us first formally define the Vertex Cover $[k]$ problem.

Vertex Cover $[k]$
Input: A graph $G = (V, E)$ and an integer $k \geq 1$.
Parameter: The size of the solution k.
Question: Is there a set $S \subseteq V$ of size at most k such that $G[V \setminus S]$ does not contain an edge (i.e. is an independent set)?

The vertex cover can be interpreted as a sort of reachability problem, where we place 'stations' on vertices such that all edges are incident on at least one station. This means the vertex cover gives us some measure of the denseness of a graph, that is, a graph with a small vertex cover is likely to contain very high degree vertices. Next to these interpretations, vertex cover has been extensively studied in the literature in interest of its complexity. Vertex Cover $[k]$ is known to be FPT, and has also been extensively researched to improve the running time of these FPT algorithms. See Table 3 for an overview of the running times for algorithms for Vertex Cover $[k]$. Furthermore, the problem of finding a vertex cover has also received attention in the streaming setting. Two interesting results in this regard are the one pass, $\widetilde{\mathcal{O}}(k^2)$ memory kernel algorithm by Chitnis *et al.* [18], and the $\mathcal{O}(2^k)$ passes, $\widetilde{\mathcal{O}}(k)$ memory branching algorithm by Chitnis and Cormode [17]. These results are already very promising, but we can wonder whether we can improve upon them, or find an algorithm with a different pass/memory trade-off. To this end, it interesting to consider the following theorem given by Abboud *et al.* [1]

Theorem 6.1. *[1, Thm 16] Any algorithm for the* Vertex Cover $[k]$ *problem which uses S bits of space and R passes must satisfy $RS \geq k^2$.*

6.2 Adapting existing branching algorithms

Let us focus on trying to improve the number of passes for the $\mathcal{O}(2^k)$ passes, $\widetilde{\mathcal{O}}(k)$ memory branching algorithm by Chitnis and Cormode [17]. Notice that $\widetilde{\mathcal{O}}(k)$ bits of memory is essentially optimal, as we generally have to save the output, which consists of $\mathcal{O}(k)$ vertices.

Consider the branching algorithm given by Balasubramanian *et al.* [6] with a running time of $\mathcal{O}((1.324718)^k k^2 + kn)$. This achieved through a two-step process: first a kernelization is executed to reduce the instance to size $\mathcal{O}(k^2)$, after which a branching algorithm is executed with a search tree size of $\mathcal{O}(1.324718^k)$. We can make this process into a streaming algorithm by ignoring the kernelization step, and executing the branching algorithm in the streaming setting, if possible. If we are able to do a single branching step in worst case $\mathcal{O}(1)$ passes, this would give us an $\mathcal{O}(1.324718^k)$ passes algorithm, as this is the size of the search tree.

Theorem 6.2. *Given a graph G as an AL stream and a solution size k, we can solve* Vertex Cover $[k]$ *using $\mathcal{O}(1.324718^k)$ passes and $\mathcal{O}(k \log n)$ bits of memory.*

Proof. We execute the branching steps as given by Balasubramanian *et al.* [6, Theorem 2] to obtain a search tree with $\mathcal{O}(1.324718^k)$ nodes. Note that we branch on at most a constant

Source	**Time Complexity**	**Year**
Mehlhorn [47]	$\mathcal{O}(2^k(n+m))$	1984
Buss and Goldsmith [12]	$\mathcal{O}(kn + 2^k k^{2k+2})$	1993
Downey and Fellows [25]	$\mathcal{O}(kn + 2^k k^2)$	1995
Balasubramanian *et al.* [6]	$\mathcal{O}(kn + 1.324718^k k^2)$	1998
Downey *et al.* [26]	$\mathcal{O}(kn + 1.31951^k k^2)$	1999
Niedermeier and Rossmanith [52]	$\mathcal{O}(kn + 1.29175^k k^2)$	1999
Stege and Fellows [61]	$\mathcal{O}(kn + \max(1.25542^k k^2, 1.2906^k k))$	1999
Chen *et al.* [15]	$\mathcal{O}(kn + 1.2852^k)$	2001
Niedermeier and Rossmanith [54]	$\mathcal{O}(kn + 1.2832^k)$	2003
Chandran and Grandoni [14]	$\mathcal{O}(kn + 1.2745^k k^4)$	2005
Chen *et al.* [16]	$\mathcal{O}(kn + 1.2738^k)$	2010

Table 3: An overview of the improvements on the Vertex Cover [k] problem. In 2000, a general technique was developed by Niedermeier and Rossmanith [53] that allows for a polynomial factor in k after the exponential factor to be removed in the running times of the algorithms. Hence, these polynomial factors are still listed for algorithms before the year 2000, but can also be removed. This does not apply to the result of Chandran and Grandoni [14], as this is an algorithm using a different technique that uses exponential space.

number of options in each step. It remains to be proven that we can execute the branching steps in each node in no more than $\mathcal{O}(1)$ passes using $\mathcal{O}(k \log n)$ bits of memory.

At any point in the branching process, denote S as the current solution set, that is, all vertices we deleted so far. In passes and checking properties we can respect the deletions in S by simple enumeration of its edits, which has no effect on the number of passes or memory usage. If at any point $|S| = k$, but we want to branch, return NO for that branch. We will now look at each of the steps of the branching process and how they can be done in the streaming setting. We always execute the first possible branching step first. These are the exact branching steps as given by Balasubramanian *et al.* [6, Theorem 2].

(1) We can find a degree 1 vertex using a pass, which also gives us its neighbour, which we can add to S.

(2) We can find a degree 2 vertex x using a pass. We can check whether its neighbours y, z are connected with an edge in another pass. If this is the case, we add y and z to S.

  (a) Otherwise, we can find the neighbours of y and z using a pass (if there are more than k, we do not branch on including these neighbours), and we can branch on including either $\{y, z\}$, or $N(\{y, z\})$.

  (b) If y and z together only have a single neighbour other than x, which we found in the previous pass, we add x and this neighbour to S.

(3) We can find a vertex x of degree at least 5 (and at most k) using a pass. We branch on including x or its neighbourhood.

(4) We can find a vertex x of degree 3 and its neighbours v_1, v_2, v_3 using a pass.

  (a) We can check whether there is a single edge between v_1, v_2, v_3 (say v_1v_2) using a pass. We branch on including $\{v_1, v_2, v_3\}$ or $N(v_3)$ (which we can find using a pass).

(b) We can check for a common neighbour y of say v_1 and v_2 in a single pass (the AL model gives us this information at y). We branch on including $\{v_1, v_2, v_3\}$ or $\{x, y\}$.

(c) We can check whether one of v_1, v_2, v_3 has at least three neighbours other than x using a pass, say this is v_1. We branch on including $\{v_1, v_2, v_3\}$, $N(v_1)$, or v_1 and $N(\{v_2, v_3\})$ (which we can find using a pass).

(d) Now each of v_1, v_2, v_3 has exactly two private neighbours excluding x. Name these $v_4, v_5; v_6, v_7; v_8, v_9$. If there is no adjacency to outside the component of vertices x and $v_1, \dots, v_9$, we can solve this locally and continue branching. Otherwise, a vertex, say v_4, has an adjacency to a vertex v_{10}. Then we branch on including $\{v_1, v_2, v_3\}$, or $\{x, v_4, v_5\}$, or $N(\{v_2, v_3, v_4, v_5\})$ (which we can find using a pass).

(5) Every vertex is now of degree 4. We pick some vertex x and its neighbours v_1, v_2, v_3, v_4 using a pass.

(a) There is an edge between v_1, v_2, v_3, v_4, say v_1v_2, which we can find using a pass. We branch on including $\{v_1, v_2, v_3, v_4\}$, or $N(v_3)$, or v_3 and $N(v_4)$ (which we can find using a pass).

(b) We can check whether three vertices (say v_1, v_2, v_3) share a common neighbour y other than x using a pass. We branch on including $\{v_1, v_2, v_3, v_4\}$ or $\{x, y\}$.

(c) Now there is a pair of vertices that in total have five neighbours other than x, which we can identify using a pass. Say this pair is v_1 and v_3. We branch on including $\{v_1, v_2, v_3, v_4\}$, or $N(v_2)$, or v_2 and $N(v_4)$, or $\{v_2, v_4\}$ and $N(\{v_1, v_3\})$ (which we can find using a pass).

We can see that we only need $\mathcal{O}(1)$ passes for these branching operations. To be able to branch when returning out of recursion in the branching procedure, it looks like we need to remember entire neighbourhoods, which can have up to size $\mathcal{O}(k)$. This would lead to an $\widetilde{\mathcal{O}}(k^2)$ memory complexity if this happens in all nodes in the branching tree. However, instead of remembering the entire neighbourhood, we can also remember what vertices we want the neighbourhood of, which is a constant amount, and use a pass to find it when returning out of recursion. This increases the number of passes by a constant factor, and makes sure that the memory use stays at $\mathcal{O}(k \log n)$ bits, as used by S. Therefore, the theorem follows. $\square$

Next, we will consider the branching algorithm given by Niedermeier and Rossmanith [52]. They show a $\mathcal{O}(kn + 1.29175^k k^2)$ running time algorithm for Vertex Cover $[k]$, relying on an extensive branch case-by-case analysis. We can hope to do the same as in Theorem 6.2, where we can implement the branching process using $\mathcal{O}(1)$ passes and so obtain a number of passes dependant on the size of the search tree alone. However, the branching rules of Niedermeier and Rossmanith are more complicated to implement with few passes.

To give a more self-contained argument, we will summarize the branching steps of Niedermeier and Rossmanith [52] below. Note that we omit details and explanations, the aim is to give a good idea of the branching steps performed. For each of these steps, we assume that we apply them from top to bottom, that is, at step i we assume that steps 1 to $i-1$ do not apply to the graph.

Step 1. If there is a vertex x with degree 1, we branch on including $N(x)$.

Step 2. If there is a vertex x with degree 6 or more, we branch on including x or $N(x)$.

Step 3. If there is a vertex of degree 2 we do the following. If the graph is 2-regular, we have a single cycle and can find an optimal vertex cover in linear time. Otherwise, assume that x is a degree-2 vertex with neighbours a, b such that a has degree ≥ 3.

Case 1. There is an edge between a and b, or a and b have a common degree-2 neighbour other than x. We include $\{a, b\}$ in the vertex cover.

Case 2. If $|N(a) \cup N(b)| \geq 4$, then we branch on including $\{a, b\}$ or $N(a) \cup N(b)$ in the vertex cover.

Case 3. If a and b have exactly one common neighbour y other than x, we branch on including $N(y)$ or $N(a)$.

Case 4. Finally, a and b have two common neighbours other than x, call them y and z. We branch on including $\{x, y, z\}$ or $N(y)$.

Step 4. If the graph is regular, choose some vertex x with maximum degree and branch on including x or $N(x)$.

Step 5. If there is a vertex of degree 3, let us call this vertex x with neighbours a, b, c. We have the following cases.

Case 1. If x is part of at least one triangle (say $\{x, a, b\}$), then we can branch on including $N(x)$ or $N(c)$.

Case 2. If x is part of at least two bridges (a bridge is where two neighbours of x have a common neighbour), where y and z are the vertices on these bridges that are not a, b, c, x, then we branch on including $N(x)$ or $\{x, y, z\}$.

Case 3. If x is part of exactly one bridge, say $\{x, a, b, y\}$, and at least one of a and b has degree 3, say a, then we branch on including $N(x)$ or $N(a)$.

Case 4. Assume the same case as Case 3, but both a and b have degree at least 4. We branch on including $N(x)$, $N(a)$, or $\{a, x, N(b), N(c)\}$.

Case 5. Now assume that there is no vertex of degree 3 that is contained in a bridge or triangle.

Case 5.1. If there is a vertex x of degree 3 with neighbours a, b, c, two of which have degree 4 (say a and b), we branch on including $N(x)$, $\{x, N(a), N(b)\}$, $\{x, a, N(b), N(c)\}$, or $\{x, b, N(a), N(c)\}$.

Case 5.2. We can now assume there is at least one vertex of degree 3 with exactly one neighbour of degree 4 or 5. We have the following cases.

Case 5.2.1. Let us assume there is no cycle of length 5 such that: (1) each vertex on the cycle has degree 3 and (2) there is a vertex on the cycle that has a neighbour with degree at least 4. Now, we assume we have some path of degree-3 vertices a_i, of which at least one has a neighbour with degree at least 4, say a_3. Niedermeier and Rossmanith give a figure to illustrate this case, see Figure 5. This figure is not complete: the degrees of the vertices b_i can differ, and these b_i vertices may be identical to vertices elsewhere in the figure. The branching is case-dependent, but roughly comes down to branching on the following options: $\{a_2, b_3, a_4\}$, $\{a_1, b_2, a_3, b_4, a_5\}$, $\{a_1, b_2, N(b_3), N(b_4), b_5, a_6\}$, or $\{a_0, b_1, N(b_2), N(b_3), b_4, a_5\}$.

Case 5.2.2. Finally, we assume there is a cycle of length 5 of degree-3 vertices where at least one vertex has a neighbour of degree at least 4. Call the cycle $a_0, \ldots, a_4$ with neighbours $b_0, \ldots, b_4$ outside the cycle. We branch on including $\{a_1, b_2, a_3\}$, $\{a_0, b_1, a_2, b_3, a_4\}$, $\{a_0, b_1, N(b_2), N(b_3), b_4\}$, or $\{b_0, N(b_1), N(b_2), b_3, a_4\}$.

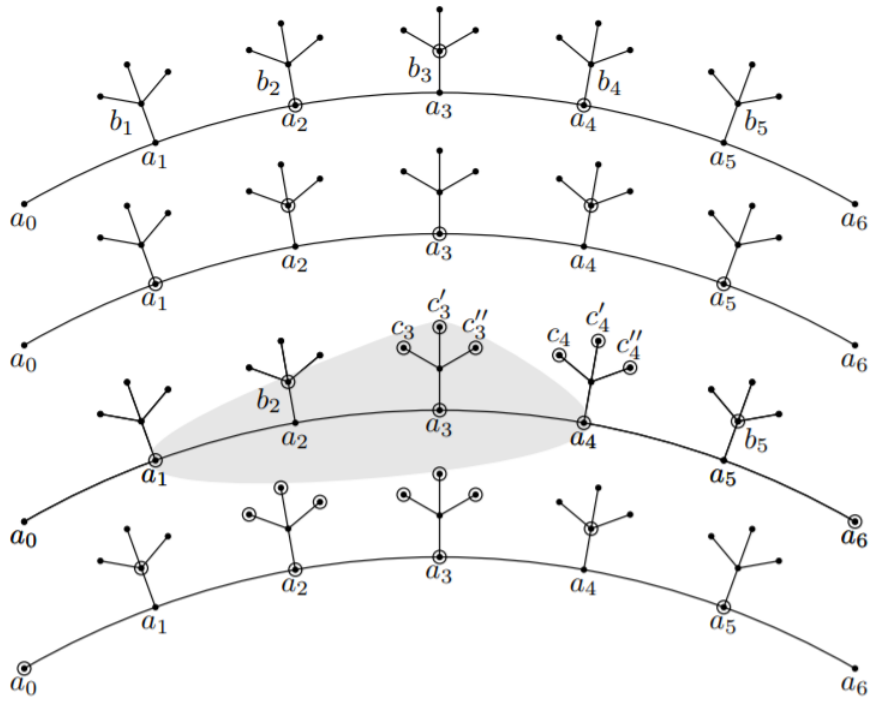

Figure 5: Image taken from [52], illustration of the branching of Case 5.2.1.

Step 6. Otherwise, there is a vertex of degree 4, and all vertices are either degree 4 or 5. We have the following cases.

Case 1. Assume there is a vertex x of degree 4 that is part of a triangle (say $\{a, b, x\}$) and has a neighbour y of degree 5. If $a, b \neq y$, and $c \neq a, b, y$ is the other neighbour of y, then we branch on including $N(x)$, $N(y)$, or $\{x, y, N(c)\}$. If, on the other hand, say $a = y$, and the other two neighbours of x, say c, d, are not connected by an edge (otherwise, let them be a and b in the previous branching), we branch on including $N(x)$, $N(y)$, or $\{x, c, N(d)\}$.

Case 2. Now assume there is no vertex of degree 4 with has both a neighbour y of degree 5 and is part of two bridges that do not include y. Now choose some vertex x of degree 4 with a neighbour a of degree 5, and let b, c, d be the other neighbours of x. We branch on including $N(a)$, $N(x)$, $\{x, a, N(b), N(d)\}$, $\{x, a, d, N(b), N(c)\}$, or $\{x, a, b, N(c), N(d)\}$.

Case 3. Now there is a vertex x of degree 4 with a neighbour y of degree 5 and x is contained in two bridges that do not include y. Call the two vertices on the bridges that are not neighbours of x a and b. We branch on including $N(x)$, $N(y)$, or $\{x, y, a, b\}$.

This concludes the branching steps of the algorithm by Niedermeier and Rossmanith [52].

When translating these branching steps to a streaming setting, there is difficulty in two factors. For one, there is some search for non-trivial structures like a cycle of five vertices consisting of degree-3 vertices where one vertex has a degree-4 neighbour. Finding such a structure in few passes is hard. Secondly, some branching rules require all previous branching rules to be non-applicable, that is, the first set needs to be applied exhaustively until the last rule can be applied. This gives us difficulty because with few passes it might be hard to check whether any of a set of rules can still apply or not. As it turns out, these issues do lead to a significant increase in the number of passes, as in each node the number of passes is not $\mathcal{O}(1)$ but some polynomial in k. However, we can limit the size of this polynomial by making use of a previously seen function, FindH (see Lemma 4.16 and Algorithms 6 and 7). FindH lends itself very well to finding small structures in a graph, as most of its complexity is dependent on the size of H, the structure it searches for. However, FindH requires a vertex cover to function, which we are searching for! Luckily, we can easily find a vertex cover of size $\leq 2k$ by finding a maximal matching in the given graph. These ideas lead to the following theorem.

Theorem 6.3. *Given a graph G as an AL stream and a solution size k, we can solve* Vertex Cover $[k]$ *using $\mathcal{O}(1.29175^k k^{\mathcal{O}(1)})$ passes and $\mathcal{O}(k \log n)$ bits of memory.*

Proof. We execute the branching steps as given by Niedermeier and Rossmanith [52] to obtain a search tree with $\mathcal{O}(1.29175^k)$ nodes. Note that we branch on at most a constant number of options in each step. It remains to be proven that we can execute the branching steps in each node in no more than $\mathcal{O}(k^{\mathcal{O}(1)})$ passes using $\mathcal{O}(k \log n)$ bits of memory.

We assume to have a vertex cover of size $\leq 2k$ in memory. This can be obtained by using one pass to find a maximal matching by using a greedy algorithm, and including all matched vertices in the vertex cover. If there are more than $2k$ of such vertices, we return NO, as any vertex cover needs to cover at least the edges in the maximal matching. This process uses no more than $\mathcal{O}(k \log n)$ bits of memory.

Steps 1 and 2 of the branching algorithm by Niedermeier and Rossmanith branch on degree-1 or degree-≥ 6 vertices, which we can easily find and execute in at most two passes. Instead of saving $N(x)$ to branch on when x is a vertex of degree ≥ 6 we can use a pass when returning out of recursion to find it again.

For Step 3, we can check whether the graph is 2-regular in a pass, in which case the solution can be deterministically found in $\mathcal{O}(k)$ passes. Otherwise, using the vertex cover of size $\mathcal{O}(k)$, we can use $\mathcal{O}(k)$ passes to find a vertex of degree 2 with a degree-3 neighbour (for every element of the vertex cover, if it has degree 2 or 3, we use a pass to check for the 'correct' case of neighbours). All the cases in Step 3 now only require inspection of the local graph surrounding the found degree-2 vertex, which can be done in $\mathcal{O}(1)$ passes. As all neighbourhoods have a constant size (at most five vertices), we can save all vertices we branch on without increasing the memory complexity.

Step 4 asks us to check whether the graph is regular, which we can do in a single pass, and then branch on the highest degree vertex, which is trivial.

Step 5 is where difficulty comes in. First off, finding a degree-3 vertex and checking and applying one of cases 1-4 requires $\mathcal{O}(1)$ passes, as it requires only local information. However, Case 5 of Step 5 asks for cases 1-4 to be non-applicable to any of the degree-3 vertices available. This is a problem, as exhaustively checking all degree-3 vertices may require $\mathcal{O}(n)$ passes. So, to avoid this problem, we make use of FindH. We can exhaustively search for the $\mathcal{O}(1)$ many local structures that Cases 1-4 ask for, each of which has $\mathcal{O}(1)$ vertices. This can be done by calling FindH for each of these $\mathcal{O}(1)$ structures with $\mathcal{O}(1)$ vertices, which means FindH requires only $\mathcal{O}(k^{\mathcal{O}(1)})$ passes and $\mathcal{O}(k \log n)$ bits of memory. Note that this does require a slight modification of FindH, where it considers the exact degrees of vertices instead of induced degree. This change is easy to make and does not change the complexity of the function. We are ready to apply Case 5. Case 5.1 is trivially applicable in $\mathcal{O}(k)$ passes. For Case 5.2, we need to distinguish whether or not a cycle of length 5, of which each vertex has degree 3 and at least one has a neighbour of degree 4, is contained in the graph. This can be done by using FindH once again, enumerating the $\mathcal{O}(1)$ many induced structures of such a cycle appearing in the graph. Regardless of whether or not such a cycle exists, the remaining branching analysis of Case 5.2 consists of $\mathcal{O}(1)$ many, $\mathcal{O}(1)$ size, complicated structures to detect, for which we use FindH again, and so we can branch effectively. All in all, Step 5 requires $\mathcal{O}(k^{\mathcal{O}(1)})$ passes, but does not exceed $\mathcal{O}(k \log n)$ bits of memory usage.

The remaining Step 6 consists of more local structure detection, for which we can easily use FindH if necessary. There are once again $\mathcal{O}(1)$ many structures we search for with size $\mathcal{O}(1)$, so we should not exceed $\mathcal{O}(k^{\mathcal{O}(1)})$ passes or $\mathcal{O}(k \log n)$ bits of memory usage.

We conclude that in each node, we require $\mathcal{O}(k^{\mathcal{O}(1)})$ passes and do not exceed $\mathcal{O}(k \log n)$ bits of memory usage, and therefore, the theorem follows. □

We can wonder what the $\mathcal{O}(1)$ in the $\mathcal{O}(1.29175^k k^{\mathcal{O}(1)})$ passes is bounded by, as this is quite relevant to the performance of the algorithm. This constant factor is decided by the calls to FindH, namely, by the size of the structures we make FindH find. Considering the size of the structures in the analysis of Niedermeier and Rossmanith [52], it should be bounded by about 30. The structures in Figure 5 reach about this size, and the other structures should not be bigger.

6.3 On vertex cover kernels

In terms of kernelization, we already have the result by Chitnis *et al.* [18], who provide a one-pass, $\widetilde{\mathcal{O}}(k^2)$-memory kernelization algorithm in the EA streaming model. It is hard to improve upon this result. However, we can observe a similar result that might be useful in finding a more memory-efficient algorithm.

A well-known kernel for the Vertex Cover $[k]$ problem is one of Buss and Goldsmith [12], which is a kernel consisting of $\mathcal{O}(k^2)$ edges. Constructing this kernel is also simple: we find all vertices with degree bigger than k, and remove them from the graph, and decrease the parameter with the number of vertices removed. If the remaining instance has a new parameter value of k', then there is no solution if there are more than $k \cdot k'$ edges. Therefore, this provides us with a kernel consisting of $\mathcal{O}(k^2)$ edges. We can observe that we are able to easily achieve this same kernel in the AL model, as identifying the degree of a vertex is not difficult in this model. Interestingly, we do not require $\widetilde{\mathcal{O}}(k^2)$ bits of memory to achieve a stream corresponding to the kernel of $\mathcal{O}(k^2)$ edges. Let us formalize this in a theorem.

Theorem 6.4. *Given a graph G as an AL stream, we can make an AL stream corresponding to the $\mathcal{O}(k^2)$-edge kernel of Buss and Goldsmith [12] for the* Vertex Cover $[k]$ *problem using two passes and $\widetilde{\mathcal{O}}(k)$ bits of memory.*

Proof. Let G be a graph, with n vertices and m edges, given as an AL stream, and let k be the solution size parameter for the Vertex Cover $[k]$ problem. Note that we can count the degree of every vertex when it appears in the stream, as we are given all adjacencies of a vertex consecutively. Therefore, in one pass over the stream we can count the degree of every vertex, and save each vertex with a degree bigger than k in a set S, as long as $|S| \leq k$. In this same pass, we keep track of two more counters: the total number of edges in the stream m' (which is $2m$), and the number of unique edges we remove r. The second can be done by keeping a local counter which we reset at every vertex and only increment for edges towards vertices not in S, and adding this local counter to the number of edges removed when we decide to add the vertex to S. If $\frac{m'}{2} - r > k \cdot (k - |S|)$, return NO. Otherwise, make a pass over the stream, and output only those edges between vertices not in S.

The output must be an AL stream, as we only remove edges from an AL stream to produce it. Let is also be clear that we use two passes over the stream.

The set S takes $\widetilde{\mathcal{O}}(k)$ bits of memory, as finding more than k vertices will result in returning NO. Counting the total number of edges takes $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ bits of space, and other counters are the same size or smaller, of which there are a constant amount. Therefore, the memory usage of this procedure is $\widetilde{\mathcal{O}}(k)$ bits.

The behaviour of this procedure is equivalent of the kernelization algorithm of Buss and Goldsmith [12], as it finds exactly those vertices with degree higher than k, and 'removes' them by adding them to S and ignoring edges incident to them in the output. Checking the instance size is done correctly, as the new parameter k' is equivalent to $k - |S|$, and the number of remaining edges is equal to $m - r$, which is $\frac{m'}{2} - r$. r is counted correctly because we only count unique edges by ignoring those towards vertices already in S. Therefore, this kernelization procedure is correct. □

Theorem 6.4 can prove useful if we can reduce the size of an $\mathcal{O}(k^2)$-edge kernel further, using only $\widetilde{\mathcal{O}}(k)$ bits of memory, if it is given as an AL stream. Applying Theorem 6.4 then essentially gives us a way to convert the original graph stream into the asked form by only increasing the number of passes by a factor 2 (we have to apply Theorem 6.4 every time the other procedure uses a pass if we want to be truly memory-efficient).

Interestingly, Chen *et al.* [15] show a way to convert the kernel of Buss and Goldsmith into a $2k$-vertex kernel for Vertex Cover $[k]$, using a theorem by Nemhauser and Trotter [51]. We will adapt this method in the streaming setting. The kernel conversion is done by converting the $\mathcal{O}(k^2)$ edges kernel into a bipartite graph, in which we find a minimum vertex cover using a maximum matching (see for example [11, Page 74, Theorem 5.3]). We can find a maximum matching in this restricted setting, as we will show that in this setting a DFS to find augmenting paths is possible in a polynomial number of passes in k and using only $\widetilde{\mathcal{O}}(k)$ bits of memory. The minimum vertex cover we find gives us the sets stated in the theorem by Nemhauser and Trotter [51], as indicated by the constructive proof of the same theorem by Bar-Yehuda and Even [7]. Lastly, we use these sets found to give the $2k$ kernel in the streaming setting as indicated by Chen *et al.* [15]. We formalize these steps in a few lemmas and theorems.

Lemma 6.5. *Given a graph G as a stream in model AL or EA, we can produce a stream in the same model corresponding to the Phase 1 bipartite graph of [7, Algorithm NT] using two passes and $\mathcal{O}(1)$ bits of memory.*

Proof. Given a graph $G = (V, E)$, Phase 1 of [7, Algorithm NT] asks for the bipartite graph B with vertex sets V, V' and edges E_B such that $V' = \{v' \mid v \in V\}$ and $E_B = \{xy' \mid xy \in E\}$. This is essentially two copies of all vertices and each edge in the original graph makes two edges, between the corresponding (original,copy)-pairs.

The process of creating a stream corresponding to B is quite simple: first we use a pass and, for every edge xy, we output xy', and then we use another pass and, for every edge xy, we output $x'y$. If the input is an EA stream, then the output must be as well, as there are no requirements. If the input stream is an AL stream, the output must be an AL stream too, as we are consistent in which copy we address. That is, the output AL stream first reveals all vertices in V and then all those in V'. All adjacencies of these vertices are present in the stream, as all the adjacencies were present in the input stream.

We can see that this uses two passes and $\mathcal{O}(1)$ bits of memory. It is trivial that the bipartite graph B is constructed correctly, as for every edge xy we output the edges xy' and $x'y$. □

Before we continue to find the maximum matching in such a graph B produced by Lemma 6.5, we need a few observations to restrict the size of the matching we want to find. The output sets

C_0 and V_0 of [7, Algorithm NT] are used by Chen *et al.* [15] to produce a $2k$ kernel. The sets C_0 and V_0 output by [7, Algorithm NT] are such that C_0 together with a minimum vertex cover on $G[V_0]$ forms a minimum vertex cover for G, and any minimum vertex cover for $G[V_0]$ includes at least $|V_0|/2$ vertices. From the conversion of these sets to a $2k$ kernel by Chen *et al.* [15], we can conclude that it must be that $|V_0| \leq 2k - 2|C_0|$ (as this shows the kernel size). But then it must also be that $|V_0| + |C_0| \leq 2k$, and, as V_0 and C_0 together include all vertices in the found minimum vertex cover in B (which has the same size as the maximum matching in B, see [11, Page 74, Theorem 5.3]). We can conclude that the maximum matching in B consists of at most $4k = \mathcal{O}(k)$ edges, and otherwise we can return NO.

Theorem 6.6. *Given a bipartite graph B as an AL stream, with $\mathcal{O}(k^2)$ edges, we can find a maximum matching of size at most $\mathcal{O}(k)$ using $\mathcal{O}(k^2)$ passes and $\widetilde{\mathcal{O}}(k)$ bits of memory. For the models EA and VA this can be done in $\mathcal{O}(k^3)$ passes.*

Proof. We first use a pass to find a maximal matching M in the graph. This can be done in a single pass because we can construct a maximal matching in a greedy manner, picking every edge that appears in the stream for which both vertices are unmatched.

Then, we iteratively find an M-augmenting path P (a path starting and ending in a unmatched vertices, alternating between edges in and not in M), and improve the matching by switching all edges on P (i.e., remove from M the edges on P in M, and add to M the edges on P not in M). Note that any such P has length $\mathcal{O}(k)$, as otherwise M would exceed size $\mathcal{O}(k)$. It is known that a matching M in a bipartite graph is maximum when there is no M-augmenting path [39]. We can also find an M-augmenting path only $\mathcal{O}(k)$ times, as the size of the matching increases by at least 1 for each M-augmenting path.

Let us now describe how we find an M-augmenting path, given some matching M of size $\mathcal{O}(k)$. We find M-augmenting paths by executing a Depth First Search (DFS) from each unmatched vertex. Note that we alternate between traversing edges in M and not in M in this search. In contrary to a normal DFS, we do not save which vertices we visited, as this would cost too much memory. Instead, we mark edges in M as visited, together with the vertex from which we started the search. If an edge $e \in M$ has been visited once in the search tree, there is no need to visit it again, as the search that visited e would have found an M-augmenting path containing e if it exists. Let us discuss the exact details on the size of the search tree and recursion.

As any M-augmenting path has length at most $\mathcal{O}(k)$, the depth of the search tree is also $\mathcal{O}(k)$. Looking at any vertex, it might have $\mathcal{O}(k^2)$ neighbours in the given bipartite graph. However, only $\mathcal{O}(k)$ of its neighbours can be in M. As visiting an unmatched vertex must end the M-augmenting path, the search tree size is only increased by visiting matched vertices. Therefore, the search comes down to the following process. From the initial unmatched vertex, we can explore to at most $\mathcal{O}(k)$ vertices (those in the matching) or any unmatched vertex which would end the search. If we explore to a matched vertex, the next step must traverse the edge in the matching to make an M-augmenting path, which is deterministic. Then we again can explore to $\mathcal{O}(k)$ matched vertices, or any unmatched vertex which would end the search. This process continues. As we only visit each matched vertex at most once, we can see that the number of vertices the search visits is bounded by $\mathcal{O}(k)$. In each node along the currently active path of the search tree, we can keep a counter with value $\leq \mathcal{O}(k^2)$ (using $\mathcal{O}(\log(k^2)) = \mathcal{O}(\log k)$ bits) to keep track of what edge we consider next. These counters take up $\mathcal{O}(k \cdot \log k) = \widetilde{\mathcal{O}}(k)$ space. In any node, if we wish to consider the next edge incident to a vertex v with a counter value x, we inspect the x-th edge incident to v in the stream. If it turns out we cannot visit that vertex (have already visited it), we can increment the counter and find the next edge to consider in the same pass (as the $(x+1)$-th edge incident to v must be later in the stream than the x-th edge incident to v). Therefore, finding the next edge to visit in the search only takes a single pass. Notice that we return to nodes in the search tree at most $\mathcal{O}(k)$ times in total, because only visiting matched vertices can result in a 'failed' search recursion. So, a search that visits all matched vertices uses $\mathcal{O}(k)$ passes, and this is the maximum number of passes for a single search.

As with any search tree branching algorithm, we pass the part of the M-augmenting path found so far to the children of a node such that the memory complexity is not increased. We start our search at most once from each unmatched vertex that has at least one edge (for which we can keep another counter to keep track), which means we do at most $\mathcal{O}(k^2)$ searches. However, for each of these searches we start from different vertices, we still keep saved the set of visited matched vertices. If a search from a vertex visits a matched vertex and does not find an M-augmenting path, then neither will a search from a different vertex by visiting that matched vertex again. In particular, this is because the graph is bipartite. This would indicate that we need to use $\mathcal{O}(k^2)$ passes, at least one for every vertex with an edge. However, in the AL model, if we consider the x-th vertex, and in the pass we use for it, we do no successful visit to a vertex (all adjacencies are matched and already visited), then in the same pass we can consider the $(x+1)$-th vertex, because all edges incident to the $(x+1)$-th vertex in the stream appear later than the edges incident to the x-th vertex in the stream. Hence, in the AL model, over all $\mathcal{O}(k^2)$ searches, we only use $\mathcal{O}(k)$ passes, because only actually visiting vertices increases the number of passes, and we can only visit $\mathcal{O}(k)$ vertices in total. In the EA and VA models, we require at least one pass for each vertex we want to start searching from, and so the total number of passes is $\mathcal{O}(k^2)$.

We conclude that with $\mathcal{O}(k)$ passes in the AL model, and $\mathcal{O}(k^2)$ passes in the EA/VA models, and $\widetilde{\mathcal{O}}(k)$ bits of memory we can execute a DFS to find an M-augmenting path (if it exists).

As mentioned, we can search for an M-augmenting path only $\mathcal{O}(k)$ times, as the existence of more M-augmenting paths would result in returning NO. Therefore, we can find a maximum matching in B using $\mathcal{O}(k^2)$ passes and $\widetilde{\mathcal{O}}(k)$ bits of memory in the AL model. In the EA or VA models, we require $\mathcal{O}(k^3)$ passes to accomplish this. □

This result is particularly interesting, as there is strong evidence that, in general, a DFS cannot be done in logarithmic space [59].

Next, we show how to convert such a maximum matching into a minimum vertex cover for B, as asked by [7, Algorithm NT].

Lemma 6.7. *Given a bipartite graph B as an AL stream and a maximum matching M of size $\mathcal{O}(k)$, we can find a minimum vertex cover X for B with $|X| = |M|$, using $\mathcal{O}(k)$ passes and $\widetilde{\mathcal{O}}(k)$ bits of memory. For the models EA and VA, this takes $\mathcal{O}(k^2)$ passes.*

Proof. We adapt a theorem by Bondy and Murty [11, Page 74, Theorem 5.3] to the streaming setting to achieve this lemma. Let us shortly go over what [11, Page 74, Theorem 5.3] entails. If our bipartite graph has vertex sets V, V' and a maximum matching M, then we can find a minimum vertex cover X with $|X| = |M|$ in the following manner. Denote all unmatched vertices in V with U, and let $Z \subseteq V \cup V'$ be the set of vertices connected to U with an M-alternating path (a path such that edges in M and not in M alternate). If $S = Z \cap V$ and $T = Z \cap V'$, then X is given by $X = (V \setminus S) \cup T$.

As $T \subseteq X$ and $|X| = |M|$, we can find and save T by executing a DFS procedure just like in Theorem 6.6, without exceeding $\widetilde{\mathcal{O}}(k)$ bits of memory. This takes $\mathcal{O}(k)$ passes and $\widetilde{\mathcal{O}}(k)$ bits of memory. Also, $V \setminus S$ must only contain matched vertices, as $U \subseteq S$. Therefore, in the same DFS procedure to find T, we can also save for every matched vertex in V if it is reachable through an M-alternating path. Then $V \setminus S$ is simply given by all matched vertices in M for which we did not save that they were reachable. We conclude that we can find X, the minimum vertex cover such that $|X| = |M|$, in $\mathcal{O}(k^5)$ passes and $\widetilde{\mathcal{O}}(k)$ bits of memory. As Theorem 6.6 works on any of the models $\mathcal{M} \in \{\text{EA, VA, AL}\}$, this procedure also works on any of these models. □

We are now ready to combine these results into a theorem.

Theorem 6.8. *Given a graph G as an AL stream, we can produce a kernel of size $2k$ for the* Vertex Cover *$[k]$ problem using $\mathcal{O}(k^2)$ passes and $\widetilde{\mathcal{O}}(k)$ bits of memory.*

Proof. We execute Theorem 6.6 on the stream produced by applying Theorem 6.4 and then Lemma 6.5 on the input stream (we have to apply these transformations every time that we require a pass). Notice that these two applications increase the number of passes by a constant factor. On the result of Theorem 6.6 we apply Lemma 6.7 to obtain a minimum vertex cover for the specific bipartite graph B. As described by Bar-Yehuda and Even [7, Algorithm NT], the sets C_0 and V_0 can be computed from this minimum vertex cover. This computation is a simple process, where C_0 contains the vertices v for which both $v, v' \in B$ are contained in the minimum vertex cover of B, and V_0 contains the vertices v where either $v, v' \in B$ is contained in the minimum vertex cover of B, but not both. Finding C_0 and V_0 from B and its minimum vertex cover requires no passes over the stream, as they are simply given by analysing the minimum vertex cover of B. These sets C_0 and V_0 are exactly the sets in the theorem by Nemhauser and Trotter [51], and, as described by Chen *et al.* [15], we can use them to obtain a $2k$ kernel for vertex cover. The kernel is given by $G' = G[V_0]$, which we can find with a pass (we can output the kernel as a stream), and parameter $k' = k_1 - |C_0|$, where k_1 is the parameter after application of Theorem 6.4. All in all, this process takes $\mathcal{O}(k^2)$ passes and $\widetilde{\mathcal{O}}(k)$ bits of memory. The requirement of an AL stream comes from the fact that Theorem 6.4 requires an AL stream to work in a constant number of passes, and Theorem 6.6 and Lemma 6.7 require less passes in the AL model. □

Note that we can apply any algorithm for Vertex Cover $[k]$ on the kernel produced by Theorem 6.8, but this would still be preferably a streaming algorithm. This is because the kernel has $2k$ vertices, but might still have $\mathcal{O}(k^2)$ edges. Nonetheless, this can lead to an algorithm heavily outperforming e.g. the algorithm of Theorem 6.3, as we may use linear memory and still remain in the optimal $\mathcal{O}(k \log n)$ memory bound, while using less passes to solve the problem.

This makes a good contribution to the pass/memory trade-off for the Vertex Cover $[k]$ problem as we have managed a kernelization algorithm with a number of passes polynomial in k while remaining space-optimal. Working in the AL model is necessary for Theorem 6.8, as Theorem 6.4 would not work without it. Counting the degrees of vertices is something the AL model trivializes in a single pass, while other models may require many more passes to accomplish this goal. Also, the DFS procedure in Theorem 6.6 and Lemma 6.7 makes clever use of the AL model to be more efficient in a single pass. It remains an open question whether similar results can be obtained in other streaming models.

7 A Different Streaming Model

We can asks ourselves whether there are different streaming models with potential use and improvement over the standard streaming models. We have seen differences in streaming models, mainly when comparing the versatile Adjacency List model to the less informative Vertex Arrival or Edge Arrival models. This difference suggests that there might be streaming models which could provide even more information leading to better and more efficient algorithms.

Finding such streaming models, and problems on which they are effective, can prove difficult, however. It seems that often different streaming models only lead to slight improvements in running time, not significant differences that change the complexity entirely. Nonetheless, let us suggest a short list of potential models which include a varying degree of information in their streams.

- A Depth First Search (DFS) or Breadth First Search (BFS) model, where the edges arrive in the order in which a DFS or BFS explores them, starting from an arbitrary fixed vertex.
- An AL or VA streaming model, but augmented such that the vertices arrive in order of their degree (highest to lowest, or lowest to highest).
- An AL or VA streaming model, but we first get the vertices contained in a minimum vertex cover of the graph, after which we get the other vertices in arbitrary order.
- A Hamiltonian Path/Cycle model, where, given that one exists, the edges arrive in order of a Hamiltonian path or cycle (after which we get the other edges). Alternatively, an Eulerian Path model, where the edges arrive in order of an Eulerian path.

Similar other models can be thought of, although it should always be remembered that demanding more information of the stream might severely increase the runtime complexity of the side producing the stream.

Let us discuss the potential uses of the suggested streaming models.

A DFS or BFS model can be useful in the regard that other structural properties of a graph can be derived from a DFS or BFS traversal. An example of this is that a DFS traversal is actually a tree-depth decomposition of the graph, and we will make use of this property in the rest of this section.

In a model where the vertices arrive in order of their degree, finding the highest or lowest degree vertex is trivialized. However, more potential use can be in algorithms that traverse or branch exactly in this order of arrival, where the degree of the vertex impacts the decisions made by the algorithm.

Interesting about the model where we first get the vertices contained in the vertex cover, is that the VA model starts to behave like the AL model when we have seen the entire vertex cover. This is because any vertex outside the vertex cover only has adjacencies in the vertex cover, and all those vertices must have arrived before it. Therefore, where the VA model normally does not give a guarantee on whether or not we see all the adjacencies of a vertex, we actually have this guarantee for all vertices not in the vertex cover. This model also gives us the vertex cover of the input graph without extra work, which is potentially useful.

If the stream first provides a Hamiltonian path or cycle, we know a lot about the reachability of vertices in the graph. The Eulerian path model is especially interesting, as we get the information that the order of edges in the stream make up a connected path that we 'walk along' throughout the pass. These models may be useful in very specific scenarios, perhaps when we wish to identify what type of graph the stream describes.

That being said, an interesting result can be obtained with a DFS streaming model for Edge Dominating Set $[k]$.

7.1 Edge Dominating Set

Let us first introduce the problem of Edge Dominating Set $[k]$. Here, we call two edges *adjacent* when they share a vertex v as an endpoint.

Edge Dominating Set $[k]$
Input: A graph $G = (V, E)$ and an integer $k \geq 1$.
Parameter: The size of the solution k.
Question: Is there a set $S \subseteq E$ of size at most k such that every edge is either in S or adjacent to an edge in S (i.e., the edges in S dominate all edges)?

In the streaming setting, Edge Dominating Set $[k]$ has already received some attention. This is mainly through the results of Fafianie and Kratsch [28], who show a lower bound of $m - \mathcal{O}(1)$ bits for a single pass algorithm, and give a kernel of size $\mathcal{O}(k^3 \log k)$ constructible in two passes using $\mathcal{O}(k^3 \log n)$ bits of memory.

7.2 Relating Edge Dominating Set, DFS, and tree-depth

We can make an interesting observation regarding Edge Dominating Set $[k]$ in combination with a depth first search model: any path the DFS traverses can only be of length $\mathcal{O}(k)$ for a solution to exist for Edge Dominating Set $[k]$. Interestingly, any depth first search traversal over a graph gives a tree-depth decomposition of the graph. We can use these two observations in conjunction to get the following lemma.

Lemma 7.1. *Given a graph G as a DFS stream, in a single pass over the stream and using $\mathcal{O}(k \log n)$ bits of memory, we can produce a stream corresponding to a tree-depth decomposition of G of depth at most $4k$ or conclude that there exists no solution to* Edge Dominating Set *$[k]$.*

Proof. Let $G = (V, E)$ be a graph given as a DFS stream.

Let us first argue that any DFS traversal corresponds to a tree-depth decomposition of the graph, as claimed in [58]. Remember that a tree-depth decomposition is a tree T with the vertices of V such that any pair $v, w \in V : vw \in E$ has an ancestor-descendant relationship in T. We argue that the edges traversed by the DFS to visit previously unvisited vertices make up the edges of a tree-depth decomposition T of G. Consider any edge $vw \in E$ with vertices $v, w \in V$. One of v, w will be explored earlier than the other by the DFS, say this is v. As we consider a DFS, w must be explored before we return out of recursion to an ancestor of v, as at least the edge vw can be traversed to explore w. Therefore, v and w have an ancestral-descendant relationship in T. This for any edge $e \in E$, and so indeed a tree-depth decomposition is given by the DFS traversal.

Let us now argue that the tree-depth decomposition of G must have depth at most $4k$ if there is a solution for Edge Dominating Set $[k]$. Consider a simple path of length x. In any solution to Edge Dominating Set $[k]$, every edge must either be in the solution, or adjacent to an edge in the solution. Looking at this path, picking any edge in E to be in a solution for Edge Dominating Set $[k]$ can dominate at most four edges on this path (if this edge is incident on two non-adjacent vertices of the path). Therefore, for a solution of size k to exists, x must be at most $4k$. If the depth of the DFS is more than $4k$, then there is a simple path of length more than $4k$, and so we can return NO. Otherwise, the depth of the tree-depth decomposition is at most $4k$.

Lastly, we should argue that we can produce a stream corresponding to the tree-depth decomposition of depth at most $4k$ using only one pass and $\mathcal{O}(k \log n)$ bits of memory, or decide that there is no solution to Edge Dominating Set $[k]$. In the pass over the DFS stream, we save the set of parents of the current vertex in T, including the current vertex we explore. We output every edge that is not between two saved vertices. This is exactly the set of edges of the tree-depth decomposition, as we proved previously that the edges traversed by the DFS to visit previously unvisited vertices make up the edges of a tree-depth decomposition T of G. We never visit an already-forgotten vertex v, as that would mean the DFS could traverse this edge when exploring v itself, which means we have already visited both vertices. We have already seen that the set of vertices we save cannot exceed $4k$ elements, as otherwise we can return NO.

The exact workings are now as follows: If the DFS stream contains an edge vw, where only v is the saved set, we do three things. Firstly, we remove all vertices from the set that were added later than v (which we can keep track of with an index per element, or by ordering the set). Then add w to the set, and output vw as part of the tree-depth decomposition. If the DFS stream contains an edge vw, where both v and w are in the set, we ignore it and move to the next edge. If the saved set exceeds size $4k$ we return NO.

This completes the proof of the lemma. $\square$

Lemma 7.1 illustrates the use of the DFS streaming model in conjunction with the Edge Dominating Set $[k]$ problem, as we can view a pass over the stream as either a pass over all the edges of the graph, or use Lemma 7.1 and view the pass as a pass over the tree-depth decomposition of depth at most $4k$. This means we will be able to use an algorithm parameterized by tree-depth to find an algorithm for Edge Dominating Set $[k]$ in the DFS streaming model.

7.3 A tree-depth algorithm

In this section, we will give an algorithm for Edge Dominating Set parameterized by tree-depth, which is an algorithm that counts the number of solutions of all possible sizes. Using Lemma 7.1 we can then change the parameter to solution size, and adapt the algorithm to the streaming setting for the DFS streaming model.

Previous work on space-efficient tree-depth algorithms can be found in [33, 36, 50, 56]. The algorithms given by these authors make use of either, or both, of the following two techniques: they make some sort of transformation to be more efficient in the saved information, or they make use of counting certain structures. The algorithm we will give is inspired by a tree-depth algorithm for Dominating Set parameterized by tree-depth, given by Pilipczuk and Wrochna [56] and further explained by Ambags [5]. This algorithm will also make use of a transformation by evaluating polynomials, and count the number of solutions instead of finding solutions directly. This will allow us to be space-efficient.

We adapt the same notation and concepts as in [5], which we will shortly go over below.

We will work on a tree-depth decomposition of the graph, for which we require some notation. Given the tree-depth decomposition T of a graph $G = (V, E)$, let $tail(v)$ denote the set of all ancestors of v, and similarly, let $tree(v)$ be the set of all descendants of v. Analogously, let $tail[v]$ and $tree[v]$ be the sets of ancestors and descendants including v itself, respectively. In the algorithm, we will work with a set of labels Σ, and labelling functions $\phi : tail(v) \to \Sigma$. When considering some labelling function $\phi : tail(v) \to \Sigma$, we will use $\phi[v \to L]$ for $v \in V(T), L \in \Sigma$, to extend the mapping ϕ with a new (key,value)-pair, that is, v is assigned the label L. We will denote $\phi^{-1}(L)$ to denote the set $\phi^{-1}(L) = \{v \in V(T) \mid \phi(v) = L\}$. Similarly, we will denote $\psi : tail[v] \to \Sigma$ to denote the function where v has received a label. The same syntax is used for ψ as for ϕ.

Intuitively, we will work top-down on the tree-depth decomposition, exhaustively considering the different possible labellings for each vertex, and then working bottom-up in returning the number of possible solutions for each labelling and combining them such that we get the total number of solutions of each possible size.

We will first define the labels we will use. The set Σ of labels will be $\Sigma = \{T, F, D\}$. We interpret these labels as taken, forbidden, and dominated, respectively.

- A vertex with label T means that at least one edge incident to this vertex will be in the solution. Hence, we can assume all edges incident to a vertex labelled T are dominated, and may freely be chosen to be included in a solution.

- A vertex with label F means that no edge incident to this vertex is in the solution, and we also do not care whether or not the edges incident on this vertex are dominated.

- A vertex with label D means that no edge incident to this vertex is in the solution, but we require that all edges incident to this vertex must be dominated in a solution.

Let us formally define the problem of which we will compute the answer.

Definition 7.2. For a graph $G = (V, E)$ with disjoint vertex sets $X, Y \subseteq V$, and a function $\phi : X \to \Sigma$, we define $EDS(G, \phi, X, Y)$ to be the problem of finding the collection of all sets $y \subseteq E(G[Y \cup \phi^{-1}(T)])$ such that:

1. each edge in $E(G[Y])$ is either contained in y or adjacent to an edge in y (i.e., they share a vertex as endpoint), and
2. $N_G(\phi^{-1}(D)) \cap (\phi^{-1}(D) \cup \phi^{-1}(F) \cup A) = \emptyset$, where A is the set of vertices in Y that have no incident edge in y.

The second demand can be interpreted as having to dominate all edges incident on vertices marked D, or at least make sure we dominate all edges of D that are part of the current scope. When $X = \emptyset$, notice that this problem asks for exactly all valid solutions to Edge Dominating Set.

In the algorithm, when considering some vertex v, the sets X, Y in Definition 7.2 will consist of $tail(v)$ and $tree[v]$, respectively. We will also not return the sets y exactly, but count them instead. To be memory efficient, we associate the different solution sizes with a polynomial defined as follows.

Definition 7.3. For a graph $G = (V, E)$ with disjoint vertex sets $X, Y \subseteq V$, and a function $\phi : X \to \Sigma$, to the set of all solutions S to the problem $EDS(G, \phi, X, Y)$ we associate a polynomial $P_{\phi,X,Y}$ that is defined as $P_{\phi,X,Y} = \sum_{0 \le k \le |Y|^2 + |X| \cdot |Y|} a_k \cdot x^k$ for $a_k, k \in \mathbb{N}_0$, where $a_k = |\{y \mid y \in EDS(G, \phi, X, Y) \wedge |y| = k\}|$.

Notice that, for a problem $EDS(G, \phi, X, Y)$, the maximum possible size of a solution y is given by $|Y|^2 + |X| \cdot |Y|$, which corresponds to picking all edges in $G[Y]$, which can be $|Y|^2$ many, and also picking all edges between vertices in Y and those in X labelled T, which can be up to $|X| \cdot |Y|$ many.

The variable x in these polynomials does not serve a semantic purpose, but is used so that we can combine the solutions easily. The idea of these polynomials is that they group the solutions by size, and the coefficients count how many solutions of size k we have seen. To be able to combine polynomials, we need a notion of independence of subproblems, defined and proven in the following lemma.

Lemma 7.4. *For a graph G, vertex sets $X, Y, Z \subseteq V(G)$ such that there are no edges between Y and Z, and a labelling function $\phi : X \to \Sigma$, the problems $EDS(G, \phi, X, Y)$ and $EDS(G, \phi, X, Z)$ are* independent*, that is, for any sets $y \subseteq E(G[Y \cup \phi^{-1}(T)])$, $z \subseteq E(G[Z \cup \phi^{-1}(T)])$, y is a solution to $EDS(G, \phi, X, Y)$ and z is a solution to $EDS(G, \phi, X, Z)$ if and only if $y \cup z$ is a solution to $EDS(G, \phi, X, Y \cup Z)$.*

Proof. Let y, z be two arbitrary solutions to the problems $EDS(G, \phi, X, Y)$ and $EDS(G, \phi, X, Z)$, respectively. By definition, $y \subseteq E(G[Y \cup \phi^{-1}(T)])$ and $z \subseteq E(G[Z \cup \phi^{-1}(T)])$ so $y \cup z \subseteq E(G[Y \cup Z \cup \phi^{-1}(T)])$. As for y all edges in $E(G[Y])$ are either contained in y or adjacent to an edge in y, and analogously for z, and there are no edges between Y and Z, then also for $y \cup z$ all edges in $E(G[Y \cup Z])$ are either contained in $y \cup z$ or adjacent to an edge in $y \cup z$. Furthermore, since $N_G(\phi^{-1}(D)) \cap (\phi^{-1}(D) \cup \phi^{-1}(F) \cup A_y) = \emptyset$, where A_y is the set of vertices in Y that have no incident edge in y, and analogously for z, then also $N_G(\phi^{-1}(D)) \cap (\phi^{-1}(D) \cup \phi^{-1}(F) \cup A) = \emptyset$, where A is the set of vertices in $Y \cup Z$ that have no incident edge in $y \cup z$. Hence, $y \cup z$ is a valid solution for the problem $EDS(G, \phi, X, Y \cup Z)$.

Now let S be a solution to the problem $EDS(G, \phi, X, Y \cup Z)$. We can split S into edges incident on vertices of Y and edges incident on vertices of Z. Let y, z be exactly these sets, respectively. Now it must be that $y \cup z = S$, as there are no edges between Y and Z, and so all edges must be incident to either some vertex in Y or some vertex in Z, but not both. Since by definition $S \subseteq E(G[Y \cup Z \cup \phi^{-1}(T)])$, and by definition y, z do not contain edges incident to Z, Y respectively, it must be that $y \subseteq E(G[Y \cup \phi^{-1}(T)])$ and $z \subseteq E(G[Z \cup \phi^{-1}(T)])$. As we argued that $y \cup z = S$ and there are no edges incident to both a vertex in Y and in Z it must now be that all edges in $E(G[Y])$ are dominated by y and all edges in $E(G[Z])$ are dominated by z, as no edge in y can possibly dominate an edge in $E(G[Z])$ and similarly no edge in z can dominate an edge in $E(G[Y])$. Lastly, as $N_G(\phi^{-1}(D)) \cap (\phi^{-1}(D) \cup \phi^{-1}(F) \cup A_S) = \emptyset$, where A_S is the set of vertices in $Y \cup Z$ that have no incident edge in S, and exactly $y \cup z = S$, then also $N_G(\phi^{-1}(D)) \cap (\phi^{-1}(D) \cup \phi^{-1}(F) \cup A_y) = \emptyset$, where A_y is the set of vertices in Y that have no incident edge in y, and analogously for z. Therefore, y and z are valid solutions to $EDS(G, \phi, X, Y)$ and $EDS(G, \phi, X, Z)$, respectively.

We have now shown that y is a solution to $EDS(G, \phi, X, Y)$ and z is a solution to $EDS(G, \phi, X, Z)$ if and only if $y \cup z$ is a solution to $EDS(G, \phi, X, Y \cup Z)$, and so, the problems $EDS(G, \phi, X, Y)$ and $EDS(G, \phi, X, Z)$ are independent. □

Ambags [5, Lemma 1] shows that for any two independent subproblems, we can count the solutions to the combined problem correctly by multiplying the corresponding polynomials of the subproblems. This property extends to combining countably many subproblems, as seen by [5, Lemma 2]. For completeness, we formally state this lemma here.

Lemma 7.5. *(Equivalent of [5, Lemma 2]) For disjoint sets of vertices $W = \{V_0, \dots, V_n\}$, let $\{EDS(G, \phi, X, V_0), \dots, EDS(G, \phi, X, V_n)\}$ be a set of mutually independent subproblems of the problem $EDS(G, \phi, X, Y)$, each having a set of solutions represented by the polynomials $P = \{P_{\phi,X,V_0}, \dots, P_{\phi,X,V_n}\}$. The polynomial $P_{\phi,X,\bigcup_{w \in W} w}$, associated with the problem $EDS(G, \phi, X, \bigcup_{w \in W} w)$, is given by $\mathcal{P} = \Pi_{q \in P} q$, the product of all polynomials in P.*

We will see that this is a very useful property in the algorithm.

Now, we will formally define some functions used in the algorithm, which correspond to certain subproblems. These are:

- $f(v, \phi) = P_{\phi, tail(v), tree[v]}$: the polynomial associated with the problem $EDS(G, \phi, tail(v), tree[v])$.
- $g(v, \psi) = P_{\psi, tail[v], tree(v)}$: the polynomial associated with the problem $EDS(G, \psi, tail[v], tree(v))$.

Notice that the difference between $f(v, \cdot)$ and $g(v, \cdot)$ is whether or not we have assigned v a label, and so whether we are interested in the solutions of the subtree where v is the root, or interested in the combined solution of the subtrees given by the children of v.

We are now ready to define the behaviour of the algorithm in terms of the functions f, g over the tree-depth decomposition. First notice that if our tree-depth decomposition has root r, then $f(r, \varnothing)$ returns the polynomial counting all solutions to the problem over the entire graph, as $tree[r]$ is the entire tree-depth decomposition and so $G[E(tree[r])] = G$. It remains to show how we resolve the values of f and g.

Let us start with g. We get the following lemma.

Lemma 7.6. *Given a graph G with its tree-depth decomposition T, an internal vertex $v \in V(T)$, and a labelling $\psi : tail[v] \to \Sigma$ we have $g(v, \psi) = \Pi_{u \in child(v)} f(u, \psi)$.*

Proof. This proof is equivalent to [5, Lemma 4], for our problem $EDS(G, \phi, X, Y)$.

We have seen that for any arbitrary disjoint vertex sets $X, Y, Z \subseteq V(G)$, such that there are no edges between Y and Z, we have that $EDS(G, \phi, X, Y)$ and $EDS(G, \phi, X, Z)$ are independent. Now consider two arbitrary children u_0, u_1 of v in the tree-depth decomposition T, and let $X = tail[v] = tail(u_0) = tail(u_1)$, $Y = tree[u_0]$, $Z = tree[u_1]$. By the properties of a tree-depth decomposition, there are no edges between Y and Z, and so the problems $EDS(G, \psi, X, Y)$ and $EDS(G, \psi, X, Z)$ are independent. Hence, by Lemma 7.5, multiplying the polynomials associated with the subproblems $EDS(G, \psi, tail(u), tree[u])$ for each $u \in child(v)$ results in the polynomial associated with the subproblem $EDS(G, \psi, tail[v], \bigcup_{u \in child(v)} tree[u])$, which is exactly $EDS(G, \psi, tail[v], tree(v))$, as desired by $g(v, \psi)$. □

This resolves the value of g for all vertices in the tree-depth decomposition, except for leaf vertices. Let us resolve these next.

Lemma 7.7. *Given a graph G with its tree-depth decomposition T, a leaf vertex $v \in V(T)$, and a labelling $\psi : tail[v] \to \Sigma$ we have*

$$\begin{aligned} g(v, \psi) &= 1 \quad \text{if } \forall u \in \psi^{-1}(D), w \in tail[v] : uw \in E(G) \implies w \in \psi^{-1}(T), \text{ and} \\ g(v, \psi) &= 0 \quad \text{otherwise.} \end{aligned}$$

Proof. Since the subtree given by $tree(v)$ is empty, and so the graph induced by the vertices contains no edges, we trivially meet the requirement that all edges must be dominated. What remains to be checked is whether or not the edges incident to the vertices marked as D are dominated by the labelling in the path to the root. We see that $g(v, \psi)$ gets value 1 exactly when every vertex in $\psi^{-1}(D)$ has only adjacencies labelled T. This is correct, as the empty set then is a solution to the problem. Otherwise, there is no solution and the value of $g(v, \psi)$ is 0. □

What remains is to resolve the value of f for all vertices. Note that there is no need to distinguish between an internal and leaf vertex for f, as we can simply assign a label to the current vertex and call g, which handles the leaf and internal vertices correctly.

Lemma 7.8. *For a graph G with tree-depth decomposition T, a vertex $v \in V(T)$, and a labelling $\phi : tail(v) \to \Sigma$ we have*

$$f(v, \phi) = g(v, \phi[v \to D]) + g(v, \phi[v \to T]) - g(v, \phi[v \to F]) + \sum_{1 \le i \le j} x^i \cdot \binom{j}{i} \cdot g(v, \phi[v \to T]),$$

where $j = |N_G(v) \cap \phi^{-1}(T)|$, the number of edges between v and vertices marked T by ϕ.

Proof. The transition between f and g requires f to decide on a label for v to be able to call on g. Naturally, because we wish to count all solutions, we have to exhaustively try the different options of labelling v and including edges in the solution that lead to different solutions.

In the case where we decide that v should not have an incident edge in the solution, we assign it the label D. The subproblems demand that edges incident on vertices labeled D are counted correctly, so no further work is required to count the solutions where v is labelled D.

In the case where we decide that v should have an incident edge in the solution, we will assign it the label T. However, this is not enough to count the solutions correctly. Actually, when labelling v, we have to decide on exactly what edges between v and vertices higher in the tree-depth decomposition labelled T we include in the solution. If we choose to include no edges to vertices higher in the tree, we have to make sure that we only count the solutions where v will have an incident edge from a vertex lower in the tree. To this end, we count exactly the solutions where v gets the label T, but we subtract the solutions where v does not get an incident edge in the solution, which is given by labelling v with F. This is also why we never check whether or not the edges of vertices labelled F are dominated, as it only matters that no edge incident to a vertex labelled F is picked in a solution.

If we do choose to include at least one edge to a vertex higher in the tree, we have to count all possibilities. The number of edges we can include is bounded by $j = |N_G(v) \cap \phi^{-1}(T)|$, as these are the only edges that are 'legal' to include. Including i edges then gives us $\binom{j}{i}$ options for choosing the edges, and adds i edges to the solution, which is why we multiply with x^i. For every solution returned by the subproblem we have then have $\binom{j}{i}$ options for including i edges to make a new solution, and so multiplying the polynomial given by g with $\binom{j}{i}$ is the correct approach. We do this for every viable i, which is why we sum over these options.

These options have exhausted the possibilities we have with regard to the vertex v: either it has one or more edges incident to it in the solution, or it does not. As such, $f(v, \phi)$ is given by the sum of all these choices, i.e., adding the polynomials corresponding to the choices together. □

As Ambags [5] and Pilipczuk and Wrochna [56] mention, if the graph consists of multiple connected components, and we have multiple disconnected tree-depth decompositions, we can multiply the respective results for calling $f(r, \varnothing)$ on the roots r of each of the disjoint trees. We conclude that we now have a polynomial P consisting of terms $a_k x^k$ where a_k is the number of solutions of size k, and so we can solve the Edge Dominating Set problem.

What remains is to analyse the time and space complexity of this algorithm.

Lemma 7.9. *(Adaptation of [5, Lemma 7]) Given a graph $G = (V, E)$ and its tree-depth decomposition T of depth s, we can compute the sequence $Q = \{q_0, q_1, \ldots, q_m\}$ such that G contains exactly q_k distinct edge dominating sets of size k for $0 \leq k \leq m$ using $\mathcal{O}(3^s \cdot poly(n))$ time and $\mathcal{O}(s \cdot m^2)$ bits of space.*

Proof. The subproblem for a vertex in the tree-depth decomposition is always defined in terms of its direct children (or given by a constant for a leaf vertex), and so we never visit the same vertex twice in the same call stack. We also never need to compute the values of $f(v, \phi)$ or $g(v, \phi)$ more than once for the same pair (v, ϕ), since f only makes three unique calls (the value of the call $g(v, \phi[v \to T])$ in $f(v, \phi)$ can immediately be used for all $\leq j + 1$ occurrences), each of which assigns a unique label to v, and g only depends on its children. Hence, the maximum number of calls to f, g is bounded by the number of unique pairs (v, ϕ). Our alphabet Σ has three unique labels, so for any v there are $3^{|tail[v]|}$ different functions ϕ, which is bounded by 3^s. In total, we have $\mathcal{O}(3^s \cdot n)$ calls to the functions f, g.

To resolve the value of $f(v, \phi)$ given the values of its subproblems, we need to do $\mathcal{O}(s)$ operations on three polynomials of degree at most m. To resolve the value of $g(v, \phi)$ we may need to multiply a linear number of polynomials of degree at most m. The coefficients of these polynomials may have values up to 2^m, the total number of subsets of a collection of size m. Just like Ambags, we abstract from the exact implementation and consider just the number of operations.

To resolve the values of f, g, we need to add and multiply a linear number of polynomials. The resulting polynomial has degree at most m, which means we require $\mathcal{O}(nm)$ operations, which each take polynomial time. We also might need to inspect the graph, checking at most $\mathcal{O}(s^2) = \mathcal{O}(n^2)$ edges. This means we can resolve the value for $f(r, \varnothing)$ in time $\mathcal{O}(3^s \cdot poly(n))$.

We are required to store at most one intermediate result per call to f and one per call to g, which consists of a polynomial of degree at most m with coefficients of size at most 2^m. Storing a constant number of such polynomials takes $\mathcal{O}(m^2)$ space. The depth of the call stack is at most $2 \cdot s$, since we call both f, g once per depth in the decomposition (in a single call stack). This leads to a $\mathcal{O}(s \cdot m^2)$ space requirement. □

The space requirement of $\mathcal{O}(s \cdot m^2)$ is not ideal, and can luckily be improved. Pilipczuk and Wrochna [56] show a technique that can make the algorithm more memory-efficient. Instead of saving the entire polynomials during the computation, we evaluate the polynomials over some Galois Field in each step, such that each polynomial is a single value taking up space $\mathcal{O}(\log n)$ bits. This means we lose information, because the single value at the root does not give us the coefficients of the polynomial anymore. However, as Pilipczuk and Wrochna [56] show, if we do this same process for different values over which we evaluate the polynomial, and for different primes for the modular arithmetic used, we can extract the original polynomial from which these values were computed. This means we can still find the number of solutions of each size, while being more memory-efficient. This procedure does come at the cost of increasing the number of passes by a *poly*(n) factor, as it has to make multiple calls to the algorithm. We state the following theorem by Pilipczuk and Wrochna that we will apply to our algorithm.

Theorem 7.10. *([56, Theorem 30]) Let* $P(x) = \sum_{i=0}^{n} q_i x^i$ *be a polynomial over one variable* x*, of degree at most* n *and with integer coefficients satisfying* $0 \leq q_i \leq 2^n$*, for* $i = 0, \ldots, n$*. Suppose that given a prime number* $p \leq 2n+2$ *and* $a \in \mathbb{F}_p$*, the value of* $(P(a) \mod p)$ *can be computed in time* T *and* S *space. Then given* $k \in \{0, \ldots, n\}$*, the value* q_k *can be computed in* $\mathcal{O}(T \cdot poly(n))$ *time and* $\mathcal{O}(S + \log n)$ *space.*

Theorem 7.10 asks for the time and space complexity of computing $(P(a) \mod p)$, where p is a given prime and $a \in \mathbb{F}_p$. We can do this by executing the tree-depth algorithm of Lemma 7.9, where we evaluate each polynomial P to $(P(a) \mod p)$ in each step. This does not change the time complexity of the algorithm, so $T = \mathcal{O}(3^s \cdot poly(n))$. The memory complexity goes down, as for each element in the tail we need not save an entire polynomial, but instead only an evaluated value, which means the memory complexity is $S = \mathcal{O}(s \cdot \log n)$ bits.

The following theorem now follows from applying Theorem 7.10 to Lemma 7.9 where we substitute m for n in Theorem 7.10, with the knowledge that $m = \mathcal{O}(n^2)$, $T = \mathcal{O}(3^s \cdot poly(n))$, and $S = \mathcal{O}(s \cdot \log n)$.

Theorem 7.11. *Given a graph* $G = (V, E)$ *and its tree-depth decomposition* T *of depth* s*, we can compute the sequence* $Q = \{q_0, q_1, \ldots, q_m\}$ *such that* G *contains exactly* q_k *distinct edge dominating sets of size* k *for* $0 \leq k \leq m$ *using* $\mathcal{O}(3^s \cdot poly(n))$ *time and* $\mathcal{O}(s \cdot \log n)$ *bits of space.*

We have to note that the change from Lemma 7.9 to Theorem 7.11 does not come without downsides, as the time complexity is increased with some polynomial in n, but as we already had an unknown polynomial in n in the running time of Lemma 7.9, this is unnoticeable.

7.3.1 Adapting the algorithm to streaming setting

Let us now adapt the algorithm of Theorem 7.11 to the streaming setting. Our aim here is to find an algorithm for Edge Dominating Set $[k]$ in the DFS streaming model. Luckily, we already have almost all tools to accomplish this result.

Previously, we have seen Lemma 7.1 that allows us to view a pass over the stream as either an inspection of the graph and its edges, or an inspection of the tree-depth decomposition. We can use this on the algorithm of Theorem 7.11 to obtain the following result.

Theorem 7.12. *Given a graph* G *as a DFS stream, we can solve* Edge Dominating Set $[k]$ *using* $\mathcal{O}(3^{4k} \cdot poly(n))$ *passes and* $\mathcal{O}(k \log n)$ *bits of space.*

Proof. First, apply Lemma 7.1 to the stream every time we wish to inspect the tree-depth decomposition, which has depth at most $4k$. Applying Lemma 7.1 takes one pass to produce one pass over the tree-depth decomposition, and $\mathcal{O}(k \log n)$ bits of space.

With this in hand, we can use the algorithm of Theorem 7.11. The only question that remains is how the time complexity relates to the number of passes we make over the stream, that is, when do we need to use passes over the stream in the algorithm of Theorem 7.11.

First, it is important to notice that Theorem 7.10 can be seen as a black-box function that requires no knowledge of the structure of the graph, and therefore requires no passes over the stream. However, using Theorem 7.10 means we call upon the tree-depth decomposition algorithm an extra $poly(n)$ number of times.

We only require passes for the computation of the functions f and g in the algorithm of Lemma 7.9. Notice that both the functions f and g may require passes when computing the (partial) value, as they both require knowledge of the structure of the tree-depth decomposition or the structure of the graph G. The function f makes at most three calls, and so is returned to in the call stack only a constant number of times. It only requires a constant number of passes each time we compute a partial value (to learn of the edges in G towards other vertices marked T), which also takes at most $\mathcal{O}(k \log n)$ space to locally save this information, which we discard when computing a value of one of the g functions. The function g requires a pass each time a partial value is computed, to view its children in the tree-depth decomposition, and call on the right child for the value of f. We return to any g function a linear number of times because it might have a linear number of children in the tree-depth decomposition. If we call g on a leaf vertex in the tree-depth decomposition, it requires a constant number of passes to decide on its value. Therefore, the number of passes made by adapting Lemma 7.9 to the streaming setting is $\mathcal{O}(3^s \cdot n^2)$.

We can now combine all these factors to get that solving Edge Dominating Set $[k]$ for a graph G given as a DFS stream requires $\mathcal{O}(3^{4k} \cdot poly(n))$ passes and $\mathcal{O}(k \log n)$ bits of space. □

We have now seen an algorithm for solving Edge Dominating Set $[k]$ in the DFS model using $\mathcal{O}(3^{4k} \cdot poly(n))$ passes and $\mathcal{O}(k \log n)$ bits of space. This is in contrast to the algorithm given by Fafianie and Kratsch [28], which is a $\mathcal{O}(k^3 \log k)$-size kernel constructible in two passes using $\mathcal{O}(k^3 \log n)$ bits of memory in the EA model. It is interesting that the DFS model allowed us to construct a memory-optimal direct algorithm for Edge Dominating Set $[k]$, where the result by Fafianie and Kratsch only provided a low-pass kernel. However, this did come at a price, the number of passes is now exponential in the solution size, and includes a polynomial factor in n. Nonetheless, this result provides a very interesting pass/memory trade-off for the Edge Dominating Set $[k]$ problem.

Let us discuss the possibility of another approach to adapting the tree-depth algorithm of Lemma 7.9 to the DFS streaming model. If we analyse the process the DFS stream makes to view the tree-depth decomposition, and compare this to the behaviour of the functions in the algorithm, we might see the possibility of developing a one-pass exponential-memory algorithm. This is where we would use the single pass over the DFS stream to traverse the tree-depth decomposition, but use exponential space to save the intermediate results for each possible state combination of the tail. However, doing this would require very close and careful analysis. This is because the check that happens at a leaf cannot be done only at the leaf, as we might find edges in the stream from the leaf to other vertices in the tail, but not edges from one vertex in the tail to another in the tail. Hence, we would need to carefully adapt this check to be partially executed in every vertex along the root-leaf path. There is also some trouble in counting the number of edges from a vertex to a vertex in the tail, as the order of the stream matters greatly here, which we do not have any control over. Hence, it seems that this approach might require us to save many partially defined polynomials, which leads to a factor of n^2 in the memory use (next to the exponential factor for the number of different state combinations). Hence, it seems this approach might not be any better than simply saving the entire graph to memory. We cannot use the memory-saving approach suggested by Theorem 7.10, as we would need to be able to at least evaluate the polynomials in each step, which proves difficult when the order of the stream impacts what intermediate results we can compute. So, we leave such a low-pass exponential-memory algorithm for a possibility of future work.

8 Conclusion and Future Work

We have seen a variety of parameterized problems tackled in the streaming model, with a main focus on the Adjacency List streaming model and parameterization by vertex cover.

We have seen some different approaches to solving the Π-free Deletion [VC] problem in the Adjacency List streaming model. We adapted existing kernels by Jansen [42] and Jansen and Kroon [41] to obtain a one-pass and $\ell + 1$-passes kernelization algorithm, which demand general conditions on Π such that the kernel holds. We then took an FPT approach to find a direct algorithm to solve the Π-free Deletion [VC] problem, in hopes of being more memory efficient. This resulted in a separate algorithm for Cluster Vertex Deletion [VC] using $\mathcal{O}(2^K K^2)$ passes and an optimal $\mathcal{O}(K \log n)$ bits of memory. The first step to generalizing this result came through an algorithm called FindH, and gave us an algorithm for H-free Deletion [VC], using $\mathcal{O}(2^K h^{K+2} K^h h!)$ or alternatively $\mathcal{O}(2^K h^{K+2} K! h!)$ passes and $\mathcal{O}((K+h^2)\log n)$ bits of space. Using this algorithm for H-free Deletion [VC], we found and algorithm for Π-free Deletion [VC] using $\mathcal{O}(q \cdot 2^K \cdot c_\Pi^K \cdot K^{K+2} \cdot K! \cdot (c_\Pi K)!)$ passes and $\widetilde{\mathcal{O}}((c_\Pi K)^2 + q \cdot (c_\Pi + 1)K)$ space, where q is the size of a specific subset of Π. We then discussed some alternative algorithms for when we do not want to have Π in memory, but instead use some oracle to learn information about Π. Afterwards, we took a sidestep to a subproblem of Π-free Deletion [VC], Odd Cycle Transversal [VC], for which we found an algorithm using $\mathcal{O}(3^K)$ passes and $\mathcal{O}(K \log n)$ bits of memory. To complement the upper bound results, we gave lower bound results for Π-free Deletion and H-free Deletion with varying conditions on the streaming model and parameter used.

We have also seen approaches for solving the Π-free Edge Editing [VC, ℓ] and Π-free Editing [VC, ℓ] problems, which required to be parameterized by both the vertex cover size and the solution size. In this context, we first discussed a branching algorithm for Cluster Editing [VC, ℓ] which we improved using existing techniques to an algorithm using $\mathcal{O}(2.27^\ell \cdot K^2)$ passes and $\mathcal{O}((K+\ell)\log n)$ bits of memory in the Adjacency List model. We then discussed kernels for Cluster Editing [VC, ℓ], where we adapted a 6ℓ kernel by Jiong Guo [35] which worked with critical cliques to the streaming setting, and gave a $2K+\ell$ kernel using an interesting observation. We then moved on to the more general Π-free Editing [VC,ℓ], for which we gave an algorithm using $\mathcal{O}(|\Pi| \cdot \nu^{2\ell+1}(K+\ell)^\nu \nu!)$ passes and $\mathcal{O}((K+\ell\nu^2)\log n + |\Pi|\nu^2)$ bits of memory, where ν is the maximum size of a graph in Π, which was inspired by an algorithm by Cai [13]. We also discussed some options when it is preferable to have Π not be explicitly present in memory.

Because of our great use of vertex cover, we moved on to find algorithms for the Vertex Cover [k] problem. We started with exploring existing branching algorithms and adapting them to the streaming setting, resulting in an algorithm using $\mathcal{O}(1.324718^k)$ passes and $\mathcal{O}(k \log n)$ bits of memory and one using $\mathcal{O}(1.29175^k k^{\mathcal{O}(1)})$ passes and $\mathcal{O}(k \log n)$ bits of memory. These algorithms both work in the Adjacency List streaming model, and are adaptations of algorithms by Balasubramanian *et al.* [6] and Niedermeier and Rossmanith [52], respectively. We then used a chain of kernels and clever conversions to obtain a kernalization algorithm using $\mathcal{O}(k^2)$ passes and $\widetilde{\mathcal{O}}(k)$ bits of memory, resulting in a $2k$ kernel, once again for the Adjacency List model.

We have seen an approach at a new streaming model being used for EDGE DOMINATING SET [k], in particular, a Depth First Search model. This model allowed us to translate the solution size parameter into a tree-depth parameter. Then, by creating an algorithm for EDGE DOMINATING SET parameterized by tree-depth, based on a tree-depth algorithm for DOMINATING SET by Pilipczuk and Wrochna [56], we obtained an algorithm for EDGE DOMINATING SET [k] using $\mathcal{O}(3^{4k} \cdot poly(n))$ passes and $\mathcal{O}(k \log n)$ bits of space in the Depth First Search model.

8.1 Future Work

The following list of open questions might prove interesting for future research.

- Are there other structural parameters like vertex cover that can lead to new results for graph problems in the streaming setting? Examples might include treewidth, pathwidth, or more on tree-depth.
- Can lower bounds be found for the Π-FREE DELETION [VC] problem in the Adjacency List model, when the vertex cover is of constant size?
- Are there streaming models different from the EA, VA, AL, and DEA models that, like the DFS model, can provide new results for graph problems?
- Are there other Oracle models for which algorithms for Π-FREE DELETION or Π-FREE EDITING can be memory efficient without having to save Π explicitly?
- Are there conditions under which a maximum-flow problem can be solved in the streaming setting? As maximum-flow problems occur often, this could prove fruitful for new results.
- Is it possible to replicate the behaviour of Theorem 6.4 in a low amount of passes for other models than the AL model? This could imply a $2k$ VERTEX COVER [k] kernel for a model other than AL.
- Is it possible to develop a $\mathcal{O}(1)$-passes $\mathcal{O}(exp(k))$-memory algorithm for EDGE DOMINATING SET [k] with the DFS streaming model, inspired by Theorem 7.12?

References

[1] Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Ami Paz. Smaller cuts, higher lower bounds. *CoRR*, abs/1901.01630, 2019.

[2] Deepak Agarwal, Andrew McGregor, Jeff M. Phillips, Suresh Venkatasubramanian, and Zhengyuan Zhu. Spatial scan statistics: approximations and performance study. In Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos, editors, *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 24–33. ACM, 2006.

[3] Akanksha Agrawal, Arindam Biswas, Édouard Bonnet, Nick Brettell, Radu Curticapean, Dániel Marx, Tillmann Miltzow, Venkatesh Raman, and Saket Saurabh. Parameterized streaming algorithms for min-ones d-sat. In Arkadev Chattopadhyay and Paul Gastin, editors, *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2019, December 11-13, 2019, Bombay, India*, volume 150 of *LIPIcs*, pages 8:1–8:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[4] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 20–29. ACM, 1996.

[5] M. M. Ambags. Exact algorithms for domination problems parameterized by tree-depth using polynomial space. Master's thesis, Technische Universiteit Eindhoven, 2019.

[6] R. Balasubramanian, Michael R. Fellows, and Venkatesh Raman. An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*, 65(3):163 – 168, 1998.

[7] Reuven Bar-Yehuda and Shimon Even. A local-ratio theorem for approximating the weighted vertex cover problem. In Manfred Nagl and Jürgen Perl, editors, *Proceedings of the WG '83, International Workshop on Graphtheoretic Concepts in Computer Science, June 16-18, 1983, Haus Ohrbeck, near Osnabrück, Germany*, pages 17–28. Universitätsverlag Rudolf Trauner, Linz, 1983.

[8] Arijit Bishnu, Arijit Ghosh, Sudeshna Kolay, Gopinath Mishra, and Saket Saurabh. Fixed-parameter tractability of graph deletion problems over data streams. *CoRR*, abs/1906.05458, 2019.

[9] Arijit Bishnu, Arijit Ghosh, Gopinath Mishra, and Sandeep Sen. On the streaming complexity of fundamental geometric problems. *CoRR*, abs/1803.06875, 2018.

[10] Sebastian Böcker and Jan Baumbach. Cluster editing. In Paola Bonizzoni, Vasco Brattka, and Benedikt Löwe, editors, *The Nature of Computation. Logic, Algorithms, Applications - 9th Conference on Computability in Europe, CiE 2013, Milan, Italy, July 1-5, 2013. Proceedings*, volume 7921 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 2013.

[11] J. Adrian Bondy and Uppaluri S. R. Murty. *Graph Theory with Applications.* Macmillan Education UK, 1976.

[12] Jonathan F. Buss and Judy Goldsmith. Nondeterminism within P. *SIAM J. Comput.*, 22(3):560–572, 1993.

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

[14] L. Sunil Chandran and Fabrizio Grandoni. Refined memorization for vertex cover. *Information Processing Letters*, 93(3):125 – 131, 2005.

[15] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2):280 – 301, 2001.

[16] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40):3736 – 3756, 2010.

[17] Rajesh Chitnis and Graham Cormode. Towards a theory of parameterized streaming algorithms. In Bart M. P. Jansen and Jan Arne Telle, editors, *14th International Symposium on Parameterized and Exact Computation, IPEC 2019, September 11-13, 2019, Munich, Germany*, volume 148 of *LIPIcs*, pages 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[18] Rajesh Chitnis, Graham Cormode, Hossein Esfandiari, Mohammad Taghi Hajiaghayi, Andrew McGregor, Morteza Monemizadeh, and Sofya Vorotnikova. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1326–1344. SIAM, 2016.

[19] Rajesh Hemant Chitnis, Graham Cormode, Hossein Esfandiari, MohammadTaghi Hajiaghayi, and Morteza Monemizadeh. Brief announcement: New streaming algorithms for parameterized maximal matching & beyond. In Guy E. Blelloch and Kunal Agrawal, editors, *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 56–58. ACM, 2015.

[20] Rajesh Hemant Chitnis, Graham Cormode, Mohammad Taghi Hajiaghayi, and Morteza Monemizadeh. Parameterized streaming: Maximal matching and vertex cover. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1234–1251. SIAM, 2015.

[21] Graham Cormode, Jacques Dark, and Christian Konrad. Independent sets in vertex-arrival streams. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 45:1–45:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[22] Christophe Crespelle, Pål Grønås Drange, Fedor V. Fomin, and Petr A. Golovach. A survey of parameterized algorithms and the complexity of edge modification. *CoRR*, abs/2001.06867, 2020.

[23] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[24] Irit Dinur and Samuel Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005.

[25] Rodney G Downey and Michael R Fellows. Parameterized computational feasibility. In *Feasible mathematics II*, pages 219–244. Springer, 1995.

[26] Rodney G. Downey, Michael R. Fellows, and Ulrike Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In Ronald L. Graham, Jan Kratochvíl, Jaroslav Nesetril, and Fred S. Roberts, editors, *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future, Proceedings of a DIMACS Workshop, Stirín Castle, Czech Republic, May 19-25, 1997*, volume 49 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 49–99. DIMACS/AMS, 1997.

[27] Facebook. Facebook reports first quarter 2020 results. Press Release, 2020 (Accessed on July 3, 2020). `https://investor.fb.com/investor-news/press-release-details/2020/Facebook-Reports-First-Quarter-2020-Results/default.aspx`.

[28] Stefan Fafianie and Stefan Kratsch. Streaming kernelization. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014*, pages 275–286, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[29] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the streaming model: the value of space. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 745–754. SIAM, 2005.

[30] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[31] Michael R. Fellows, Lars Jaffke, Aliz Izabella Király, Frances A. Rosamond, and Mathias Weller. What is known about vertex cover kernelization? In Hans-Joachim Böckenhauer, Dennis Komm, and Walter Unger, editors, *Adventures Between Lower Bounds and Higher Altitudes - Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, volume 11011 of *Lecture Notes in Computer Science*, pages 330–356. Springer, 2018.

[32] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[33] Martin Fürer and Huiwen Yu. Space saving by dynamic algebraization based on tree-depth. *Theory Comput. Syst.*, 61(2):283–304, 2017.

[34] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. In Rossella Petreschi, Giuseppe Persiano, and Riccardo Silvestri, editors, *Algorithms and Complexity, 5th Italian Conference, CIAC 2003, Rome, Italy, May 28-30, 2003, Proceedings*, volume 2653 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2003.

[35] Jiong Guo. A more effective linear kernelization for cluster editing. *Theor. Comput. Sci.*, 410(8-10):718–726, 2009.

[36] Falko Hegerfeld and Stefan Kratsch. Solving connectivity problems parameterized by tree-depth in single-exponential time and polynomial space. In Christophe Paul and Markus Bläser, editors, *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, France*, volume 154 of *LIPIcs*, pages 29:1–29:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[37] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. In James M. Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, May 20-22, 1998*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 107–118. DIMACS/AMS, 1998.

[38] Danny Hermelin, Matthias Mnich, Erik Jan van Leeuwen, and Gerhard J. Woeginger. Domination when the stars are out. *ACM Trans. Algorithms*, 15(2):25:1–25:90, 2019.

[39] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[40] Falk Hüffner, Christian Komusiewicz, Hannes Moser, and Rolf Niedermeier. Fixed-parameter algorithms for cluster vertex deletion. *Theory Comput. Syst.*, 47(1):196–217, 2010.

[41] Bart M. P. Jansen and Jari J. H. de Kroon. Preprocessing vertex-deletion problems: Characterizing graph properties by low-rank adjacencies. *CoRR*, abs/2004.08818, 2020.

[42] Bart M.P. Jansen. *The power of data reduction: Kernels for fundamental graph problems.* PhD thesis, Utrecht University, 2013.

[43] Ton Kloks, Dieter Kratsch, and Haiko Müller. Finding and counting small induced subgraphs efficiently. *Inf. Process. Lett.*, 74(3-4):115–121, 2000.

[44] Donald E Knuth. Generating all n-tuples. *The art of computer programming*, 4:54–57, 2004.

[45] Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Rec.*, 43(1):9–20, 2014.

[46] Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. Better algorithms for counting triangles in data streams. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 401–411. ACM, 2016.

[47] Kurt Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, volume 2 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1984.

[48] Hannes Moser and Dimitrios M. Thilikos. Parameterized complexity of finding regular induced subgraphs. *J. Discrete Algorithms*, 7(2):181–190, 2009.

[49] J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 253–258. IEEE Computer Society, 1978.

[50] Jesper Nederlof, Michal Pilipczuk, Céline M. F. Swennenhuis, and Karol Wegrzycki. Hamiltonian cycle parameterized by treedepth in single exponential time and polynomial space. *CoRR*, abs/2002.04368, 2020.

[51] George L. Nemhauser and Leslie E. Trotter Jr. Vertex packings: Structural properties and algorithms. *Math. Program.*, 8(1):232–248, 1975.

[52] Rolf Niedermeier and Peter Rossmanith. Upper bounds for vertex cover further improved. In Christoph Meinel and Sophie Tison, editors, *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*, volume 1563 of *Lecture Notes in Computer Science*, pages 561–570. Springer, 1999.

[53] Rolf Niedermeier and Peter Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73(3-4):125–129, 2000.

[54] Rolf Niedermeier and Peter Rossmanith. On efficient fixed-parameter algorithms for weighted vertex cover. *Journal of Algorithms*, 47(2):63 – 77, 2003.

[55] Vangelis Th. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *ACM Comput. Surv.*, 29(2):171–209, 1997.

[56] Michal Pilipczuk and Marcin Wrochna. On space efficiency of algorithms working on structural decompositions of graphs. *ACM Trans. Comput. Theory*, 9(4):18:1–18:36, 2018.

[57] Bruce A. Reed, Kaleigh Smith, and Adrian Vetta. Finding odd cycle transversals. *Oper. Res. Lett.*, 32(4):299–301, 2004.

[58] Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, and Somnath Sikdar. A faster parameterized algorithm for treedepth. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 931–942. Springer, 2014.

[59] John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.

[60] Robert H. Morris Sr. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, 1978.

[61] Ulrike Stege and Michael Ralph Fellows. An improved fixed parameter tractable algorithm for vertex cover. *Technical report/Departement Informatik, ETH Zürich*, 318, 1999.

[62] Twitter. Twitter financials and metrics. Press Release, 2020 (Accessed on July 3, 2020). `https://s22.q4cdn.com/826641620/files/doc_financials/2020/q1/Q1-2020-Selected-Financials-and-Metrics.pdf`.

A Dictionary Orderings

In this section, we will elaborate in more detail the inner workings of dictionary orderings, as defined by Definition 4.14. As a reminder, these dictionaries have as a purpose to enumerate sets of elements that satisfy a certain condition, e.g. being a subset of some given set, without using too much memory. For doing exactly this, we are faced with two challenges. Firstly, we have to find a way of enumerating such that every set that satisfies the conditions is guaranteed to occur. Secondly, we cannot use too much memory, which may disallow us to save certain supersets.

Let us first ask ourselves what we can and cannot save in memory in the use cases in this thesis. To this end, let us assume we enumerate sets of vertices in a graph $G = (V, E)$ where the vertices V are numbered $1, \ldots, n$. In the cases where the algorithm is parameterized by vertex cover, we can assume that we have a subset of vertices $X \subseteq V$ in memory corresponding to the vertex cover. Because we focus on memory-optimality, we generally want to avoid saving a set of unbounded size, i.e. some set with a size relational to n. However, given that we know that the vertices are numbered $1, \ldots, n$, we might only require to save the value of n for enumeration. Enumeration is then possible because we know that every value in the (discrete) interval of $1, \ldots, n$ corresponds to a vertex. Because we generally also have the vertex cover in memory, we can also enumerate subsets of the vertex cover or subset of the graph avoiding the vertex cover, without the demand that this is some complete discrete interval of vertices.

Let us now go into ways of enumerating specific subsets such that all subsets meeting the demands are enumerated. For this, we can look towards the literature, see for example Donald Knuth's *The Art of Computer Programming* [44]. However, we will give a self contained argument here.

For this argument, let us assume we have to enumerate all subsets S of a size $\leq c$ of a certain set U. We also assume that either we can contain U in memory, or U is some complete discrete interval with maximum value n possibly excluding some set in memory. Note that all our applications of these dictionary orderings meet these demands.

If there is no minimum size, we start with the empty set. If there is a minimum size d, start with the first d elements of U. Then, whenever the NEXT function is called on a set S, we create the next set S' from S in the following manner. Assume S has k elements and that they are sorted. Take the biggest number in the set (the k-th element), and increment it, that is, take the smallest number following it that is still allowed in U, and replace the k-th element with it. If this does not exist, we first increment the $(k-1)$-th element (continuing to the $(k-2)$-th element if that is not possible, etc.) and then reset the k-th element to the value one higher than the $(k-1)$-th element. Note that in this process, the $(k-i)$-th element may only take values up to and including $(n-i)$, as otherwise some 'bigger' values in the set cannot take any value. If this means that no value can be incremented, and $k < c$, then let S' be the first $k+1$ values of U. If $k = c$ and no value can be incremented let $S' = \spadesuit$.

Note that the definition of increment can differ depending on the context. If $U = \{1, \ldots, n\}$, then an increment can be a literal increment of the number. However, if we consider some subset $X \subseteq V$ in memory that either the sets S must be contained in or not contain any elements of, we have a different definition of increment. Here, the increment can be an iteration of literal increments until the number is allowed again, checking it in each step. This way, enumeration is possible for more arbitrary subsets.

For illustration, let us give a short example. Say we want to enumerate subsets of $V = \{1, \ldots, 5\}$ that do not contain a value in the set $X = \{3\}$ and have size at most 3. We start with the initial value where $S = \emptyset$. Then, as no value can be incremented, the first value of V is added, so $S = \{1\}$. In the next call this is incremented, so $S = \{2\}$. In the next call this is incremented, but 3 is illegal so it is skipped, so $S = \{4\}$, followed by $S = \{5\}$ in the next call.

Now 5 cannot be incremented, as it is the maximum value, so we add an element to S, which gets us $S = \{1, 2\}$. Now the largest value is incremented and we skip 3, so $S = \{1, 4\}$. This is followed by $S = \{1, 5\}$. 5 cannot be incremented further, so we increment one smaller value, netting us $S = \{2, 4\}$. Note that the 5 was replaced by the value one bigger than 2, which is 4, as 3 is illegal. This process continues with the following sets for S, in order. $\{2, 5\}$, $\{4, 5\}$, $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 4, 5\}$, $\{2, 4, 5\}$, $\{1, 2, 4, 5\}$, $\spadesuit$. We can see that we have enumerated 16 different sets, which is exactly $\binom{4}{0} + \binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4}$ many. We can conclude that when $S = \spadesuit$ we indeed have seen every subset meeting the demands.

The memory use of this approach is easy to analyse. Saving the maximum number n takes $\log n$ bits, which is dominated by the sets the ordering returns, which take $\mathcal{O}(c \log n)$ bits if the set has at most c elements. If an illegal or non-interval set is in memory, its memory usage must also be accounted for. We can see that generally, the memory usage of a dictionary ordering is negligible in comparison to the memory use of the algorithms it is used in.

Let us argue the correctness of enumerating all subsets that meet the requirements. For a constant size of sets c, this process is an example of an iterative process used for example for brute forcing a combination lock with c digits, where we skip the options where a digit appears multiple times. Therefore, by the brute force nature, this process considers all sets of size c within the requirements. If all options of size c have been enumerated, the process moves on to all options of size $c+1$ (if this is allowed), and the process starts at $c = 0$, or the given minimum size. We conclude that we indeed have correctly enumerated all sets that meet the requirements when we return $\spadesuit$.