Geometric covers, graph orientations, counter games

Edvin Berglin

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Geometric covers, graph orientations, counter games

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Edvin Berglin
December 13, 2017

Abstract

Geometric Cover is a large family of NP-complete special cases of the broader Set Cover problem. Unlike the general problem, Geometric Cover involves objects that exist in a geometric setting, consequently implying that they are all restricted to obeying some inherent structure. The archetypal example is Line Cover, also known as Point-Line Cover, where a set of points in a geometric space are to be covered by placing a restricted number of lines. We present new FPT algorithms for the sub-family Curve Cover (which includes Line Cover), as well as for Hyperplane Cover restricted to $\mathbb{R}^3$ (i.e. Plane Cover), with improved time complexity compared to the previous best results. Our improvements are derived from a more careful treatment of the geometric properties of the covering objects than before, and invoking incidence bounds from pure geometry.

An orientation of an un-directed graph is a directed version of it, i.e. where every un-directed edge in the original graph has been replaced by a directed edge, incident on the same two vertices, in either direction. Graph orientations with low out-degree are desirable as the foundation of data structures with many applications. If the un-directed graph is dynamic (can be altered by some outside actor), some orientations may need to be reversed in order to maintain the low out-degree. We present a new algorithm that is simpler than earlier work, yet matches or outperforms the efficiency of these results with very few exceptions.

Counter games are a type of abstract game played over a set of counters holding values, and these values may be moved between counters according to some set of rules. Typically they are played between two players: the adversary who tries to concentrate the greatest value possible in a single counter, and the benevolent player who tries to prevent the adversary from doing so. These counter games are sometimes used as a behind-the-scenes tool for proving the efficiency of an algorithm, i.e. proving that the adversary is unable concentrate more than some specific value in a counter, also proves that the algorithm cannot perform worse than this value. We develop a new counter game with only one player (the adversary), and use it to prove the efficiency of the graph orientation algorithm.

Resumé

Geometrisk Dækning er en stor familie af NP-fuldstændige specialtilfælde af det mere generelle Mængdedækningsproblem. I modsætning til det generelle problem involverer Geometrisk Dækning objekter der findes i en geometrisk ramme, hvilket medfører at de alle har en bestemt struktur. Det ærketypiske eksempel er Linjedækning, også kendt som Punkt-linje-dækning, hvor en mængde punkter i et geometrisk rum skal dækkes ved at placere et begrænset antal linjer. Vi præsenterer en ny FPT-algoritme for underfamilien Kurvedækning (som inkluderer Linjedækning) samt Hyperplandækning begrænset til $\mathbb{R}^3$ (dvs. Plandækning), med forbedret tidskompleksitet sammenlignet med de tidligere bedste resultater. Vores forbedringer skyldes en mere omhyggelig behandling af de geometriske egenskaber af dækobjekterne end førhen samt en anvendelse af incidensgrænser fra ren geometri.

En orientering af en uorienteret graf er en orienteret udgave af den, dvs. hvor hver uorienteret kant i den oprindelige graf er erstattet af en orienteret kant mellem de samme to kanter i en af de to mulige retninger. Graforienteringer med lav udgrad er ønskelige da de danner fundament for datastrukturer med mange anvendelser. Hvis den uorienterede graf er dynamisk (dvs. kan ændres af en udenforstående aktør), kan det være at man er nødt til at vende orienteringen af visse kanter for at bibeholde den lave udgrad. Vi præsenterer en ny algoritme der er simplere end tidligere resultater, men som opnår eller forbedrer effektiviteten af disse resultater med ganske få undtagelser.

Tællespil er en type af abstrakte spil som spilles hen over en mængde tællere der har værdier, og disse værdier kan flyttes mellem tællere i henhold til nogle regler. Typisk spilles de mellem to spillere: modstanderen, der forsøger at koncentrere den højest mulige værdi i én tæller, og den godhjertede spiller, som forsøger at forhindre modstanderen i at gøre sådan. Disse tællespil bruges nogle gange under overfladen som et værktøj til at bevise effektiviteten af en algoritme, dvs. hvis man kan bevise at modstanderen ikke er i stand til at koncentrere højere end en bestemt værdi i en tæller, har man også bevist at algoritmen ikke kan have en dårligere præstation end denne værdi. Vi udvikler et nyt tællespil med kun én spiller (modstanderen) og bruger det til at bevise effektiviteten af graforienteringsproblemet.

Preface

This thesis is in two parts. Part 1 starts with giving an over-arching introduction to the field of theoretical computer science and the relevant branches of mathematics, as well as why computer science is important in a broader perspective. Published research papers are given as-is in part 2. It features two papers; the first is *Applications of incidence bounds in point covering problems* [1] by Peyman Afshani, Edvin Berglin, Ingo van Duijn and Jesper Sindahl Nielsen, published in the 32rd International Symposium on Computational Geometry (SoCG 2016). The second is *A simple greedy algorithm for dynamic graph orientation* [3] by Edvin Berglin and Gerth Stølting Brodal, published in the 28th International Symposium on Algorithms and Computation (ISAAC 2017). Part 2 also presents some additional original research.

In general, part 2 is very technical and mathematically heavy in nature, and might be a difficult read for those not familiar with mathematical research. The latter half of Part 1 instead presents the results in a more reader friendly manner and is intended to be better suited for readers who wish to know our findings and "what they mean", without necessarily understanding how those conclusions were made. Part 1 therefore makes very light use of mathematical notation and arguments, so as to be accessible to readers without higher mathematical training. It will sometimes make statements that may not be *perfectly accurate* in order to better paint the overall picture. It features a comprehensive discussion on practical implications of our results, using real-world examples and comparisons to other research, which is something that tends to be skimmed over in research publications.

Acknowledgments

It is my understanding of a 'typical' PhD thesis – if there is such a thing – that the Acknowledgments section will feature a lengthy list of people whose influence and contributions may range from major and direct to rather minor and indirect. While I could make such a list, I feel it would only serve to diminish the colossal impact the following made. Thank you Trine Ji Holmgaard, Ingelise Lauritsen, and my parents Ingrid and Börje. Your efforts pulled me through what was by far the worst period in my life. In the most literal sense possible, this thesis would never have been written were it not for you.

Edvin Berglin,
Aarhus, December 13, 2017.

Contents

Part I

Overview

Chapter 1

Introduction

1.1 Layman's introduction to TCS

In theoretical computer science (*TCS*), we study the fundamental abilities and limitations of computers. Although one might be quick to parse the name TCS as a theoretical science about computers, it is perhaps more accurate to read it as the science of theoretical computers. I.e., the term *computer* should not be understood only as a physical device powered by electricity, but as *any* entity which can carry out calculations. It is generally of no concern whether this entity can only be built using not-yet-available technology, or even if it breaks the laws of physics and can thus never exist as anything more than an idea. Whatever the case, we develop a model of a computer with basic operations that should capture the entity's capabilities, and then draw mathematical conclusions about what that model can and cannot do. Two things should be pointed out: firstly, the computer (model) is *abstract* in the sense that it exists only "on paper" and does not require any physicality to it. Secondly, and conversely, the model is *concrete* in the sense that its set of basic operations must be very specific and detailed; if they are not, we cannot draw any accurate conclusions thereof.

For purposes of this thesis, TCS can be considered in very rough terms to have three layers[1], comprising computability theory at the top, then computational complexity theory, and algorithms and data structures at the bottom. Computability theory investigates which problems can be solved at all; it was proven in the 1930s [38] that there are some problems which, despite appearing fairly innocuous at first sight, cannot be correctly solved by *any* imaginable computer. These discoveries largely coincided with the so-called *foundational crisis of mathematics* and surprised many contemporary researchers, even to the point of outright disbelief.

Beyond telling us whether a problem can be solved or not, it seems a natural question to ask *how efficiently* it can be done. This is the point of inquiry of computational complexity theory, wherein we measure any type of resource consumption by a computer as it performs its task. The two most common type of resources that get considered are *time* – when we ask a computer to perform some computation, how long must we wait before we get the answer? – respectively *space* – how much disk space (or similar) does the computer require to answer the question, without running the risk of crashing?

But in addition to these classical dimensions of efficiency, one may in principle consider any sort of abstract cost, and one such example features prominently in Chapter 4. However, we are crucially not interested in measuring this resource consumption in absolute terms. In effect, it is of little interest for a theorist to measure the time in actual seconds; such numbers are

[1]Let us stress that this view is narrow and incomplete, and purposely neglects to account for a large host of fields that fall within (or intersect with) TCS.

immediately deprecated as soon as a team of engineers puts their latest creation on the market. Instead we measure the *growth* of the consumption as the computer solves a larger problem of the same 'type'. In other words if a computer takes, say, a minute to answer a question on a specific set of data (*input*), how long will it need to answer the same question on another input that is twice as large? In many cases we can hope to get the answer after two minutes or close thereto. Except under very specific circumstances, this is the best we can hope for. But computational complexity theory tells us that, depending on which sort of question was asked, a more accurate estimate of the time it takes to answer this second question might be in the hundreds of millions of years or worse. Since an average person may be disinclined to wait for that long, it is prudent to be able to anticipate which type of questions will exhibit this rather severe slowdown. Computational complexity theory is precisely the attempt to order problems into broad categories, according to how quickly their resource consumption grows as functions of the input size.

The most famous unresolved question within computational complexity theory is called the *P vs. NP problem*, where P and NP are two such categories. We give a more thorough description later in this thesis, but the informal gist of it can be stated as: does there exist a problem such that a computer can quickly *verify* a claimed solution, but still needs an impractically long time to *solve* the same question on its own? Coincidentally, we know the existence of very many computational problems (important in the sciences and, curiously, some situations in everyday life) that seem to exactly fit this description: it is easy to verify purported solutions, but so far they have resisted all attempts at quickly finding these solutions. The P vs. NP problem is a strong contender for the position as the most important open problem in the entirety of mathematics for this reason.

Growth of the computational resource costs is usually described using *Big-Oh notation*, $\mathcal{O}(f(n))$, to essentially say that as the *input size* grows to n, the cost does not grow past the value of the function $f(n)$. This way we say that the growth is polynomial if $f(n) \leq n^c$ for some constant c, exponential if $f(n) \leq c^n$, and so on. The aforementioned P is exactly the class of all problems with polynomial-growth time cost. This is where algorithms and data structures enter the picture: while computational complexity theory might say that a problem requires exponential time, there is still a dramatic difference between solving it in $\mathcal{O}(2^n)$ time compared to, say, $\mathcal{O}(1.2^n)$ time. Similarly, there will be a very noticeable improvement between solving a problem in $\mathcal{O}(n^2)$ time compared to $\mathcal{O}(n^4)$, even for inputs of relatively modest size. Algorithms are therefore developed to solve any specific problem faster than was possible before, or equivalently, to be able to solve a significantly larger problem within the same time limit. The time gains that can be made from an improved algorithm are often far more significant than what can be expected from using a faster computer. We use data structures to store and query data in a way that is efficient for the application, so their purpose can be said to be two-fold: they both enable an algorithm to work even faster, and organize the data to be readily available for larger programs that respond to external events (i.e. programs that listen for further inputs after they begin running, contrary to our usual assumption that the entire input is available at the start).

To return to the earlier example of the waiting time that shot up to a hundred million years, if that computer was executing a $\mathcal{O}(2^n)$ time algorithm and we replaced it with a $\mathcal{O}(1.2^n)$ algorithm, the larger input would instead only require a little under 3 days – on the exact same computer hardware. Of course it depends on the application whether this is an acceptable waiting time or not; it can be acceptable to wait 3 days when calculating the most energy-efficient way of launching a new space probe to the distant parts of the solar system, but likely not when calling to order a taxi and the company has to decide which of their unoccupied vehicles is closest-by. In either case, the situation is no longer as grossly unmanageable as before,

and at this stage it might be worthwhile to look into finding other improvements such as a more powerful computer, a higher-skilled programmer or a better-optimized compiler. Consequently, and somewhat in spite of the statements made in the opening paragraphs, algorithmists will often have a practical outlook on their work; they want their designs running on real computers, effectivizing real software and improving the user experience of real people.

1.2 Combinatorics and geometry

Combinatorics is one of the main branches in mathematics and concerns discrete structures and their manipulations. Its conceptual prominence notwithstanding, *discrete* is a word difficult to succinctly define, and is perhaps easiest understood as the opposite of *continuous*. The two can be exemplified by the integers and real numbers, respectively, where one can speak of any integer having "closest neighbours" but real numbers having no such concept. In fact, sandwiched between any two real numbers are infinitely many other real numbers, so there necessarily cannot be a 'closest' or 'next' number. Where real numbers transition 'smoothly' and without break, there is a marked 'gap' between any two integers. Combinatorics can then be said to study the different combinations and arrangements of these structures.

It could be thought to encompass all of computer science, since computers are fundamentally working on discrete data (strings of bits) and in discrete states (separated by clock cycles). But conversely, given enough working memory a computer may itself describe the states of discrete systems

One of the most basic objects in combinatorics is the graph. Unfortunately this word suffers from describing two very different mathematical concepts and most people grow up learning *the other one*. For the purposes of this thesis, a graph is *not* a visual representation of a function or its 'curve'. A graph *is* a collection of objects which can somehow relate to each other – we refer to the objects as vertices and their relations as edges. At the risk of alienating non-contemporary readers, the author has found success explaining the concept in his personal life by referring to 'the Facebook graph', where vertices represent people (or more accurately, user accounts) and 'relating' to each other in this graph represents the state of being friends. Note that a Facebook user account does not have a particular shape, nor really a physical position – it might, however, be said to have a *relative* position, measured against other accounts but not any fixed reference point.

Shapes and positions are instead the focus of geometry, an essentially different branch of mathematics (but with some significant intersection with combinatorics) and one that enjoys a long and distinguished history dating several thousand years. Well-known examples are the Pythagorean theorem in right-angled triangles and various numerical approximations of the circle constant π.

Computational geometry is a cousin field to general geometry, and unsurprisingly deals with how to make computations within geometric settings. It is the heart of every form of computer graphics and virtual world building so, at the very least, it is of central importance to much of the modern entertainment industry. But its significance is greater than that: its algorithms also govern robotic motion, computer vision and efficient layout of components on integrated circuits ("microchips").

1.3 NP and coping with hardness

Theorists distinguish between natural and artificial computational problems; natural roughly means that 'someone else' (who is not a theoretical computer science) wants to be able to solve the problem, for whatever reason. An artificial problem is instead a computational problem

that likely has no particular value or applicability outside TCS, but which may provide insights into the field itself and enable further research.

In the youth of computer science, with the advent of rigorous and fine-grained analysis of algorithms, it was noted with some frustration that a large and diverse set of computational problems seemed to intrinsically require *much* more time to solve than others. These 'much harder' problems included many natural problems that were important to other sciences. It is a common colloquial expression to say that something "grows exponentially" or is "exponentially larger", with no precise meaning except that it is or grows to be very big. For mathematicians, exponential growth has a very precise meaning (and indeed does grow very fast), and what was noted was that all algorithmic attempts for these certain problems exhibited exponentially growing running time.

In the early 1970s came a partial explanation when it was shown that these problems *reduce* to each other – if any one of them has an algorithm that is significantly faster than exponential, then all of them do. Furthermore it was shown that one of these problems (called SAT) was expressive enough that it could capture any polynomial-time algorithm running on a so-called *non-deterministic Turing machine*, a type of computer model which possesses a far greater set of abilities than a 'classical' computer does. In other words, even though we cannot physically build a non-deterministic computer, we can simulate its execution of any polynomial-time algorithm by instead solving SAT. This set of problems solvable with polynomial-time algorithms on a non-deterministic Turing machine is what we call NP – where N is for non-deterministic and P is for polynomial. SAT and the other reducible problems are called *NP-complete*.

Taken together these two results mean that if one could solve SAT, or any other of the NP-complete problems, with a polynomial-time algorithm on a classical computer, then the extra powers of non-determinism amount to no benefit at all; the classical computer would be equally powerful to the non-deterministic one. Most researchers agree that this state of affairs appears to be unlikely, so the answer to whether we can find significantly faster algorithms to NP-complete problems seems to be a somewhat disappointing "probably not".

But these natural problems nevertheless need to be solved in reasonable time. This has led to the development of several different techniques for circumventing the inherent hardness, or 'intractability', of NP-complete problems. The most basic one is to employ heuristics, which simply are rules and ideas that "seem to work most of the time" but without any sort of guarantee. In a situation where it is not imperative to find the exact solution, and if we in fact accept the possibility of getting an answer that is outright wrong, then we can often use heuristics to construct a very fast algorithm indeed. Two immediate successor techniques are approximation and randomization. Approximation algorithms do not guarantee finding the optimal solution, but do guarantee a result whose 'quality' is not much worse than the optimal one – even without explicitly knowing the quality of that solution. Randomized algorithms instead use probability theory to guarantee a certain success chance of finding the correct solution, but with the caveat that the computation may instead fail, even spectacularly, and give any garbage answer. These two paradigms readily combine with each other: randomized approximation algorithms will, with at least some probability, produce an answer of not-too-bad quality.

Another, markedly different, approach is that of parameterized algorithms. Here we off-load the intractability of the general problem onto some parameter of the input, rather than its size. The idea springs from the observation that even though a problem might be intractable in general, it can have many special instances that are considerably more easy to solve. In fortuitous cases, we can create a family of varying 'degree of specialness' (the parameter) by which these instances can be categorized, where more special instances get progressively easier to solve. Probably the most common parameter is the maximum size of the desired solution,

since small solutions – if they exist – should be easier to detect. However, the parameter does not have to be explicitly provided, it can instead be a known structural property of the input, which can then be exploited by the algorithm. We say that the problem is *fixed-parameter tractable*, i.e. the problem is no longer intractable and can be solved efficiently when when all inputs have fixed, small-enough parameters. The class of fixed-parameter tractable problems is appropriately called FPT. It should be noted that the existence of parameterized algorithm does not change the fact that the problem was intractable in general; hence there must necessarily exist inputs where the structural parameter is large or where solutions only exists for large explicit parameters.

An important concept to parameterized algorithms is that of *kernels*. They can be considered to be the 'hard core' that remains after stripping away the easy or 'soft' parts around them. This is an evocative analogy to the anatomy of common edible fruit (where the inside seed is often called a kernel), but in our case the *soft parts* are to be understood as the parts of an input that do not contribute to it being difficult to solve – the *hard* core is the part that is computationally difficult to solve. To qualify as a kernel in this sense, the hard part of the instance must have *bounded size* measured against the parameter. So for any parameterized problem, one will first try to find a kernel in reasonable time, before trying any slower algorithm on the now hopefully smaller input. Much more can be said about kernels, but this is enough to follow the arguments in the next chapter.

Chapter 2

Results in perspective

2.1 Geometric Covers

The branch of geometry known as incidence geometry studies points of intersection between different types of geometric objects. The field has roots at least as far back as the 19th century, and has a long string of developments since. Most relevant for this thesis is the notion of *incidence bounds*: given sets of geometric objects with no fixed positions, what is the maximum number of incidences (intersections) that can be achieved by freely placing these objects in any arrangement? For our purposes, we are only interested in incidence bounds between two sets where the first set contains points and the other is a set of objects that share a 'type' out of many possibilities: lines, circles, hyperplanes, parabolas, etc. Incidence bounds for these objects are well studied, but for all theoretical developments within incidence geometry and the practical importance of the greater field of computational geometry, incidence bounds have struggled to find algorithmic use.

Geometric Cover is a large sub-family of the well-known Set Cover problem. The division between Set Cover and Geometric Cover does well to illustrate how geometric objects are more

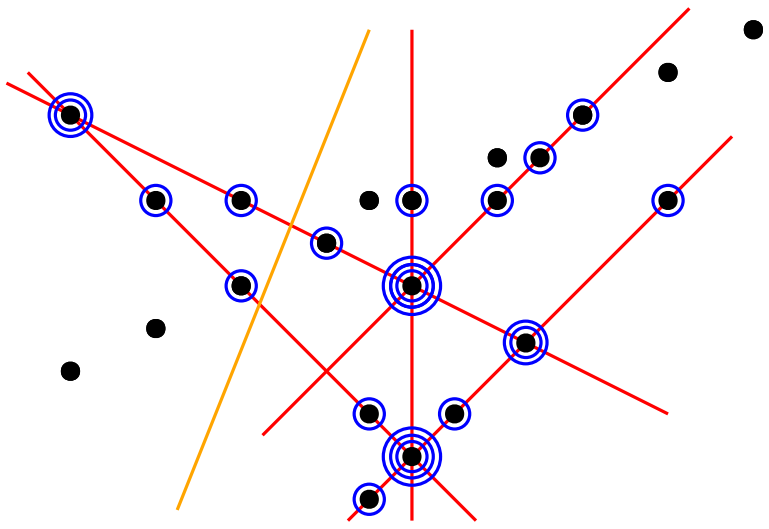

Figure 2.1: An arrangement of 22 points and 6 lines, generating 22 incidences (blue circles). More incidences are possible, for example by placing the orange line elsewhere, but the maximum possible is considerably less than 22×6.

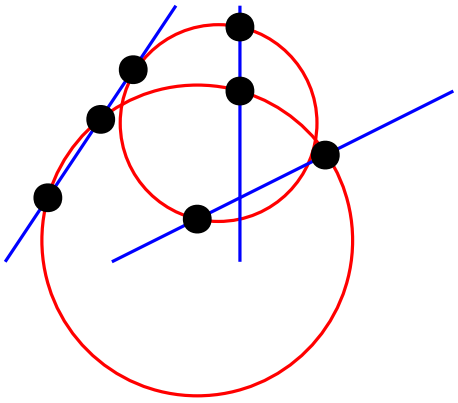

Figure 2.2: Points can be covered by different geometric objects. This small set of points requires 3 lines but only 2 circles to cover.

constrained. Say you are given a set of n elements and your task is to cover these with some type of *covering objects*. In Set Cover, these objects come from a pool of size 2^n, because any set is constructed by freely picking and choosing among the n elements. In Line Cover – the simplest case of Geometric Cover – the covering objects are much more restricted. We will only place lines that cover at least two points; let take two points p_1 and p_2 and say that the line ℓ covers them. Suppose now that there is a third point p_3 covered by ℓ. Then we immediately know that there is no line that cover p_1 and p_2 but not p_3 (likewise, no line covers p_1, p_3 but not p_2, nor p_2, p_3 but not p_1). This is a *very nice* property to have, and one that is completely absent for Set Cover. It helps explain why Geometric Cover has FPT algorithms and Set Cover does not, and similarly why one probably cannot design approximation algorithms to Set Cover that are as good as those that already exist for Geometric Cover.

There have been a handful previous papers with algorithmic improvements to Line Cover and other related Geometric Cover problems. The greatest improvement in terms of running time is due to [39], which improved the Line Cover from $\mathcal{O}\Big((k/2.2)^{2k}\Big)$ to $\mathcal{O}\Big((k/1.35)^{k}\Big)$ – the halving of the exponent from $2k$ to k has a tremendous impact. But curiously, even though the most glaring difference between Geometric Cover and Set Cover is the previously mentioned fact that geometric objects are far more constrained than combinatorial ones, previous research have neglected to really capitalize on this. The improvement due to [39] derive their improvement largely through a more careful treatment of the combinatorial properties of the problem. In the paper *Applications of incidence bounds in point covering problems* (Chapter 3), we show how to use incidence bounds as a new angle of attack against these problems. This leads to a new set of algorithms which all have a better theoretic bound on their running time than previously known work, although the theoretical improvement is not huge and they will probably not be faster in practice. However, it represents one of the first algorithmic applications of incidence bounds in general.

Intuition behind the algorithms

We give a brief intuition for how incidence bounds are employed in our algorithms, exemplified with the easily understood problem Line Cover. This problem has a well-known kernel of k^2 points, where k is the "budget" of lines we are allowed to place. Unfortunately for us, it has been proven that no better kernel than this exists.

Consider that in any arrangement of n points and m lines, the number of incidences where

(a point lies on a line) is bounded by a function $f(n, m)$. Now, suppose instead that we are interested only in lines that intersect with 'many' of the n points. Then each such line generates many incidences, but still the total number of incidences is bounded by $f(n, m)$. Hence one can necessarily only place a few such lines, *regardless* of how the points are positioned.

Now take the Line Cover problem. Suppose the task is to cover 'very many' points, using 'few' lines. Then intuitively, it is not possible to cover these many points with few lines, if each individual line covers only a few points. So at least some line must cover many points. The incidence bounds promise us that there are only a low number of ways of placing lines that cover many points. In other words, there is a small set of 'candidates' from which we can pick our necessary line – we don't know which of these lines is the correct one (and possibly we should pick more than one of them), but there are at least only a few ways to try it. Computer scientists call this *branching*: identifying a number of possible partial solutions (a line is a *partial solution* if the whole solution is made up of several lines), and systematically trying each of them in order. Each choice then leads to a different set of new possible choices, and whenever the algorithm makes the wrong choice it will typically not be discovered until after making several more choices. This is why the algorithm must be able to go back and "change its mind" from a previous decision it made, eventually exhausting all possible choices until the right one is found.

So the algorithm first tries to pick the correct line which covers many points – as stated the algorithm is deciding between only a few possible options. Furthermore, whichever line is picked, it covers (and removes!) many points, and each branch contains a significantly smaller *sub-problem* of all the points that have yet to be covered. We say that the algorithm is making good progress, as it is getting rid of a high number of points with comparatively low effort. Now the algorithm is faced with a new choice between candidates that cover "many but somewhat fewer" points, and then continuing that process between candidates that cover progressively fewer points.

Unfortunately, as we look for candidates with fewer and fewer points, the incidence bounds simultaneously get worse: there are *very many* ways of placing a line that covers only a handful of points, so consequently there are very many partial solutions to inspect. And since these lines cover a handful of points, each decision makes very little progress. The incidence bounds become so weak that continuing this process to solve the entirety of the input is too slow. Fortunately we can, at any time we wish, switch to a different algorithm if that appears better suited to solve the remaining sub-problem. And, equally fortunately, there is another algorithm whose running time is not sensitive to incidence bounds, but is instead highly sensitive the the number of points. This algorithm would be too slow, by a dramatic margin, to use on the original input – even after getting the k^2 point kernel. Things work out well, as it out that there is a 'sweet spot' – the moment that the incidence bounds become too weak for the branching algorithm, is exactly the same moment that there are few enough remaining points to run the second algorithm. See Figure 2.3 for an illustration of how these algorithms relate to each other.

Going further

Our family of Geometric Cover algorithms come with only a small theoretic improvement in running time for the best studied problems – $\mathcal{O}\Big((k/\log k)^k\Big)$ instead of $\mathcal{O}\Big((k/1.35)^k\Big)$ for Line Cover – and will probably not be more efficient in practice without using additional tricks. But it demonstrates a whole new area of applicability of incidence geometry, and it motivates a new direction of research within incidence geometry. One main reason we cannot improve our algorithm further is that there are no stronger incidence bounds we can use – the currently existing bounds are tight when making no assumptions on the structure of the objects. But the worst-case instances for number of incidences are in fact not worst-case instances for our

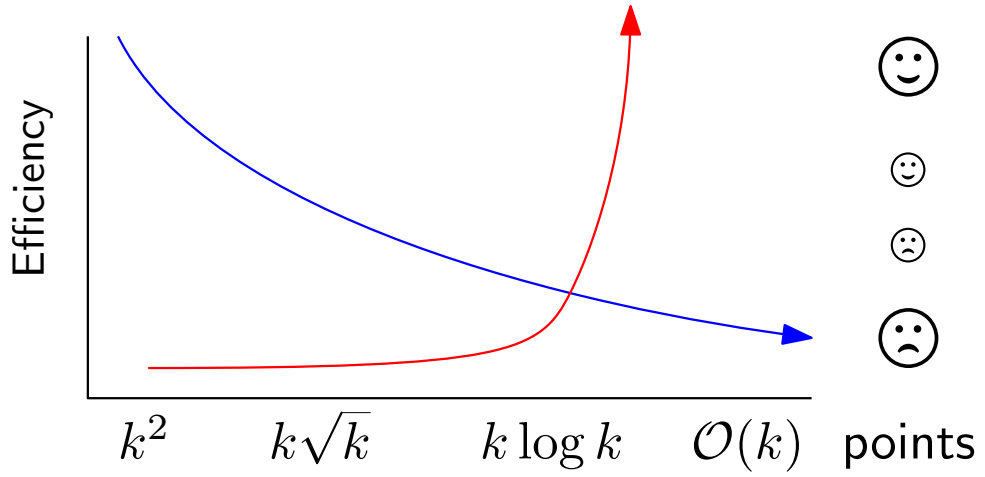

Figure 2.3: Highly simplified illustration of efficiency between the two algorithms. The 'total efficiency' corresponds to the lowest point on the curve. Switching algorithms when their curves cross, we narrowly avoid dipping down to the "large sad face" symbolizing no improvement.

algorithms. This is because these problems have kernels, where if "enough" points exist that can all be covered by a single curve, then we know for a fact that this curve *must* be in the solution, and we can immediately remove the points. This contrasts starkly with how worst-case instances for incidence bounds are constructed, which all involve placing very many points on the same curve. Clearly, the instances that generate the most incidences, are *not* the instances which are the hardest to solve.

The current bounds are very general in that they assume that both the points and the lines can be placed freely to create the maximum number of incidences. But clearly in the computational problem the points are already placed – it would be trivially easy to cover all the points with a single line if we were allowed to decide their positions. Furthermore there are arrangements that generate many incidences but are not harder for the algorithm to deal with, due to the kernel. Therefore the general bounds on the number of possible incidences over-estimate the number of guesses that the algorithm has to make. With specialized bounds which take these facts into account, it should be possible to demonstrate that our algorithms are in fact even faster than our current proofs show. Hence our work invites research into refined incidence bounds for kernelized instances. See Figure 2.4 for a simplified illustration of how such a specialized bound could improve our algorithm.

In fact, our algorithm for Plane Cover employs such a specialized bound. The tight bound for general instances is too weak to be usable with our algorithmic approach. The special bound is adapted to situations that are similar to our kernel, but not exactly the same. Hence we have some hope that the algorithm can be further improved somewhat. Additionally, this specialized bound only applies to $\mathbb{R}^3$, but the idea behind the algorithm can extend to higher dimensions if provided usable bounds.

The Curve Cover algorithms readily adapts to improved bounds: their pseudocode require only minor changes and if improved bounds become a common occurrence then the pseudocode can be made to adapt automatically. The current analysis for the time complexity is quite non-trivial but modifying it for new bounds should mostly be an exercise of plugging in the adjusted numbers where relevant; not something done in a minute, but perhaps in a day.

So to summarize, our work is the first to employ deeper facts about geometry to attack the Geometric Cover family of problems, and simultaneously one of the first to demonstrate how

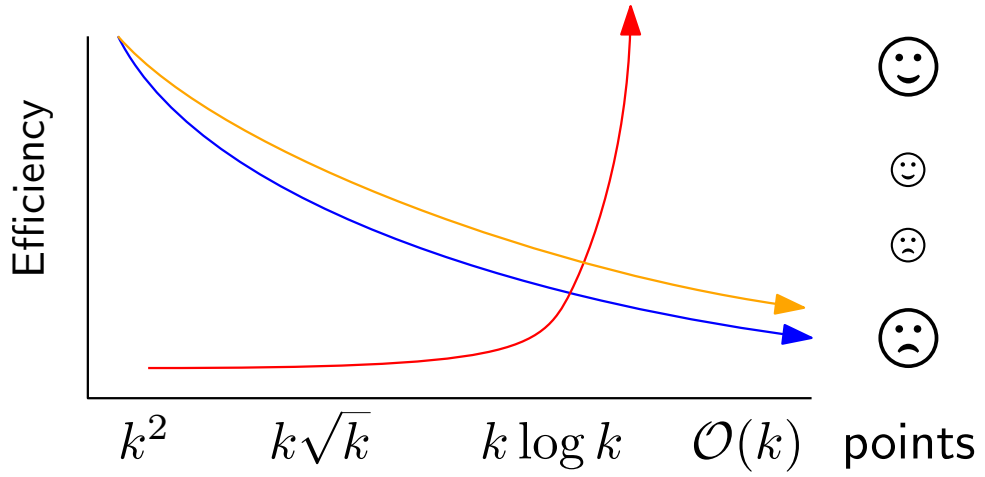

Figure 2.4: Speculative illustration of how specialized incidence bounds may improve the efficiency of the branching algorithm (from blue to orange), without modifying the algorithm itself. The new intersection point with the red curve is farther away from the "large sad face" level.

incidence bounds can be useful for algorithmic purposes. We also offer an invitation into a new direction of studying incidence bounds which would lead to further algorithmic improvements in a fairly straight-forward manner.

2.2 Dynamic graph orientation

Some graphs are very big. The example of the Facebook graph demonstrates this in a particularly forceful way. A few months before this writing, it was announced that Facebook had crossed the milestone of 2 billion user accounts [41]. It is also a very well-connected graph, as it was announced in 2016 [6] that the graph had an average *degree of separation* of less than 5: if you pick any two Facebook users in the world, then the number of steps needed to connect these two users through a chain of "friend of a friend of a friend..." is at most four, so that the fifth step reaches the destination. There can, of course, be other such chains that are much longer, and likewise it can be hard to tell chain to go through to ensure not more than five steps.

But despite its vast size and connectedness, the Facebook graph has low *degree*, i.e. every user has a relatively low number of friends. Data from 2013 [40] suggests that the median number of friends is 342, with a small but not insignificant fraction of users having more than 1,000 friends and very few indeed having more than 2,000. In graph theory this is known as a graph being *sparse*: by a large margin, most edges (friendships) that could exist in fact do not. Although Facebook does not allow a user to have more than 5,000 friends (and therefore remains relatively sparse even if every user maxes out his or her list of friends), a graph could still be said to be sparse even with vertices of very high degree as long as these vertices are very rare. So to make the example work for the below discussion, let us assume that there are a handful of Facebook users with extremely many friends, say a few million at least.

Efficient storage

The most obvious and simple way to store a graph in a computer is to use a square matrix of ones and zeros: a 'one' in the position on row x and in column y means that the users with

account numbers x and y are friends, while a 'zero' means they are not. But given the great size of the Facebook graph, the square of that number is so immense that there is simply no computer that is large enough to even come close to storing this matrix. But since we know that the graph is sparse, we know that almost all the matrix entries would be zero, so we are being very (*very*) wasteful in this storage model. A different idea would be that each account stores a list of their own friends. This is space efficient, in that we do not store any large amounts of unimportant zeroes, but we run into a different problem: those few but very popular users have lists that contain millions of other users. So if we want to ask a question of the type "are these two users friends?", which was very easy to do in the matrix, we now find ourselves forced to look trough a very long list. We could try to be clever and look in the list belonging to the 'less popular' user among the two, but this approach does not help when trying to find out if two very popular users are friends with each other.

Furthermore, the graph is not *static*: new friendships are established probably every second, as well as removed. Whichever storage model ('data structure') we decide on, it must be able to handle not just questions ('queries') but also changes ('updates'). As the above discussion suggests, we need to look to more clever data structures to have a chance of handling these demands. Fortunately, storage of graphs is a problem that has enjoyed a tremendous amount of research, including the narrower area of storing dynamic and sparse graphs. It should be noted that there are several independent ways of measuring the sparsity of a graph. *One* such way is called arboricity and in the paper *A simple greedy algorithm for dynamic graph orientation* (Chapter 4), we make a number of improvements to the state-of-the-art algorithms whose main approach is arboricity.

It should be stressed that arboricity is not always the correct sparsity measure for every type of graph and there are many other approaches that 'make sense' for this very basic problem. However, it can be difficult to compare the efficiency between two algorithms with different basic approaches; it almost needs to be done on a case-by-case basis with a specific graph at hand.

Our algorithm

The principal idea of our paper is to say that one vertex 'owns' the edge (one user owns the friendship), and including it in only that list. Furthermore, as a general rule but (crucially) not an absolute rule, the edge is owned by the vertex with lower degree (less popular user), and when the graph structure changes sufficiently, it may be necessary to reassign ownership to the other vertex. Note how this works out even for an edge between two vertices that both have very high degree: even though we need to store the edge between them somewhere, both of their lists will still be short since most of their other neighbours have lower degree. This idea is called *orienting* the graph, and is far from novel but has been utilized by several authors in previous research. The crux of the issue is that can be difficult to decide "on the fly" which vertex should own the edge, and when to reassign edges to avoid having to reassign many at the same time. It has been proven that guaranteeing the shortest possible lists requires many simultaneous reassignments, which is precisely the reason why the general is not an absolute rule. Relaxing the length requirements allows for fewer reassignments, although the exact relationship between the two quantities is still very much unknown. The various existing algorithms are therefore essentially different ways of making these reassignment decisions quickly, and to guarantee lists of acceptable length rather than shortest possible length.

The modus operandi of our new algorithm in Chapter 4 is particularly simple: it takes an edge from the vertex that owns the most edges, and reverses the ownership of this edge. Then repeat this process a set number of times, without making any decisions on a deeper level. The

majority of the paper is dedicated to proving that this simple method actually works and is competitive to previous algorithms which operate in a more complicated way.

The new algorithm in comes with a few significant strengths and improvements over previous work. First of all, our algorithm performs at least equally well as previous worst-case algorithms for almost all values of arboricity – there is only a very small range of arboricity values where our algorithm cannot match earlier work, and the difference is tiny. Conversely, there a much larger range of arboricity where our algorithm performs better. It should be noted that the existing amortized algorithms have better performance measured as an average over several updates. But critically, they have no good mechanisms for controlling when the reassignments happen, which means they are not "smoothed out" and can instead happen all at once. This results in the computer system temporarily locking up to deal with the high amount of reassignments, oftentimes for long enough to be noticed by the user, but without any reason obvious to them.

Secondly, the algorithm is trivial to implement in any normal programming language, because it does not require any complicated data structures under the hood, and it makes all its decisions almost "without even looking", i.e. without any form of contemplation about whether a particular decision might be good or not. This is known as being *greedy* in computer science lingo. Thirdly, it adapts to future research in a very interesting manner, because it can be seen as an approximation algorithm that does not perform much worse than any other algorithm. Hence, if some very clever algorithm is discovered in the future with a better length-to-reassignment balance, then our algorithm automatically performs better than what is currently known to do.

Finally, the algorithm comes with worst-case bounds and enables the programmer to pick his favorite from a wide range of balances between list lengths and reassignments, which was previously only possible using algorithms with amortized bounds. Previous worst-case algorithms instead offer only a "one size fits all" balance which may not perfectly suit every type of application. Perhaps most notably, it is the first result that can perform only a constant number of reassignments per update, i.e. the number of reassignments does not depend on the size of the graph.

Counter games

Games have a much different meaning in mathematics than in the common vocabulary, in that it does not attribute any form of entertainment (or educational, informational, etc.) value to the playing of the game. Instead, it merely describes a system with a set of rules, in which players with conflicting goals make actions – sometimes alternatingly, sometimes not – that obey the rules and which influence the shared *game state*. *Game theory* is a large field which studies how players should act in order to best reach their goals, with the knowledge that any action may have negative consequences for other players and their goals; these other players may then choose to retaliate. Game theory has exceptionally many and important applications in fields ranging from finance to evolutionary biology.

Games are also frequently used within computer science – again, we do not mean *computer games* – as tools for proving statements on certain processes. First one shows that some game is expressive enough to model the changing state of some other system, i.e. if the state of the system changes in a particular way, then the rules of the game also allow the players to replicate that change in their own setting. The rules may at the same time offer *additional powers* to the adversary, so that his ability to manipulate the game state exceed what is necessary in order to model the system state. Then one proves that, even with these extra powers, the adversary is still unable to force his way to a game state that is "too bad" due to being counter-acted by the benevolent player.

Counter games are one particular type of such game. It is played on a number of counters that hold values, and these values may be moved around between counters. For illustration, think of the counters as boxes that each hold a certain amount of an unspecified substance, and that this substance can be taken from one box and put into another. The adversary's goal is to concentrate as much as possible of the substance in any single box, and the benevolent player instead tries to make sure that every box stays relatively empty. Both players take turn in performing moves according to some rules, and these rules are not necessarily the same for both players.

In Chapter 4 we design a new counter game with only a single player, the adversary. In place of the missing opposite player are stricter rules restrictions on the moves that the adversary can perform. He cannot take from a box unless it contains almost as much as the box that contains the most. Furthermore he can only take a bounded amount of 'substance' per move, which means he cannot *empty* any box except when all boxes contain very little. As soon as the content of any box grows large enough, all boxes with significantly less content become locked and their contents are no longer redistributable. The 'largest difference' between the maximum fill of any box, and the fill level for legally removing content from another box, is called the *resolution*.

This counter game is essential to proving the out-degree guarantee for our dynamic orientation algorithm in Chapter 4, and we believe it can be used separately in further research too. In Chapter 5 we give a *lossy* extension to this counter game, in which the adversarial player must follows the same rules as above, and the added hurdle that he incurs a multiplicative loss with every move. In other words, every time he removes content from one box, he can redistribute only some fraction of that content to other boxes - the remaining moved content is lost, "sifts through his hands", and cannot be recovered.

Maximal matching

The astute reader may have detected an incongruence of words: we began Chapter 2 by describing the graph orientation work as a venture in data structures, and continued with an exposition on the need for efficient graph data structures in applications such as Facebook, but then switched to speaking of the result as an algorithm. In truth, the division between algorithms and data structures are not always clear-cut. Algorithms rely on using data structures to be efficient, but algorithms can also be used as a building block to power more complex data structures. So to put a formal label on our work, it is probably most accurately described as an algorithm which is intended as a building block to be used under the hood of various data structures.

Our work purposely leaves many implementation details unspecified, to give the programmer maximal control to balance not just list length against number of reassignments, but also the *time* it takes to update and query the structure. This way, the algorithm readily combines with standard paradigms for building data structures, such as self-balancing binary search trees, but also with more complex constructions. It really depends on the given context, in which the orientation is to be used, which implementation is preferable.

One very prominent example is called maximum matching. This is a famous problem within computer science, and often taught to novice computer science students as part of their basic curriculum. Here our Facebook example breaks away from any sort of realistic scenario, but suppose for illustration's sake that we wish to pair up every Facebook user in the world and force them to hold hands, i.e. to *match* them. It is not allowed to hold the hand of more than one other person, and two users will only agree to hold hands if they are friends, which means it might not be possible to guarantee that *every* user is paired with someone else. Still, we want to ensure that as many users as possible are paired (*maximum* matching) under these stipulations.

This task seems absurd, but the underlying principle of pairing up vertices that share an edge, and updating the matching every time the graph changes, is a basic computational problem which is used as a data structure inside a host of algorithms for more complex problems. In teaching, typically only the static case is considered, where the graph cannot change. But suppose again that the graph can see insertions and deletions of edges at any time, and that these updates cannot be predicted in advance. The task is now to quickly modify the existing pair assignments to still ensure the maximum number of users are still paired. The dynamic matching problem itself has seen a resurgence in research interest in the most recent couple of years, in no small part due to the discovery [31] that one can maintain an approximate maximum matching by using any efficient solution for the dynamic graph orientation problem. As we present new efficient orientation algorithms, we can also implicitly improve the efficiency of existing maximum matching approximation algorithms.

Part II

Publications

Chapter 3

Applications of incidence bounds in point covering problems

Peyman Afshani, Edvin Berglin, Ingo van Duijn, Jesper Sindahl Nielsen

In the *Line Cover* problem a set of n points is given and the task is to cover the points using either the minimum number of lines or at most k lines. In *Curve Cover*, a generalization of *Line Cover*, the task is to cover the points using curves with d degrees of freedom. Another generalization is the *Hyperplane Cover* problem where points in d-dimensional space are to be covered by hyperplanes. All these problems have kernels of polynomial size, where the parameter is the minimum number of lines, curves, or hyperplanes needed.

First we give a non-parameterized algorithm for both problems in $\mathcal{O}^*(2^n)$ (where the $\mathcal{O}^*(\cdot)$ notation hides polynomial factors of n) time and polynomial space, beating a previous exponential-space result. Combining this with incidence bounds similar to the famous Szemerédi-Trotter bound, we present a *Curve Cover* algorithm with running time $\mathcal{O}^*\Big((Ck/\log k)^{(d-1)k}\Big)$, where C is some constant. Our result improves the previous best times $\mathcal{O}^*\Big((k/1.35)^k\Big)$ for *Line Cover* (where $d = 2$), $\mathcal{O}^*\Big(k^{dk}\Big)$ for general *Curve Cover*, as well as a few other bounds for covering points by parabolas or conics. We also present an algorithm for *Hyperplane Cover* in $\mathbb{R}^3$ with running time $\mathcal{O}^*\Big((Ck^2/\log^{1/5} k)^k\Big)$, improving on the previous time of $\mathcal{O}^*\Big((k^2/1.3)^k\Big)$.

3.1 Introduction

In the *Line Cover* problem a set of points in $\mathbb{R}^2$ is given and the task is to cover them using either the minimum number of lines, or at most k lines where k is given as a parameter in the input. It is related to *Minimum Bend Euclidean TSP* and has been studied in connection with facility location problems [15, 30]. The *Line Cover* problem is one of the few low-dimensional geometric problems that are known to be NP-complete [30]. Furthermore *Line Cover* is APX-hard, i.e., it is NP-hard to approximate within factor $(1+\varepsilon)$ for arbitrarily small ε [26]. Although NP-hard, *Line Cover* is fixed-parameter tractable when parameterized by its solution size k so any solution that is "not too large" can be found quickly.

One generalization of the *Line Cover* problem is the *Hyperplane Cover* problem, where the task is to use the minimum number of hyperplanes to cover points in d-dimensional space. Another generalization is to cover points with algebraic curves, e.g. circles, ellipses, parabolas, or bounded degree polynomials. These can be categorized as covering points in an arbitrary dimension space using algebraic curves with d degrees of freedom and at most s pairwise intersections. We call this problem *Curve Cover*. The first parameterized algorithm that was presented for *Line Cover* runs in time $\mathcal{O}^*\left(k^{2k}\right)$ [27].[1] This algorithm generalizes to generic settings, such as *Curve Cover* and *Hyperplane Cover*, obtaining the running time $\mathcal{O}^*\left(k^{dk}\right)$ where d is the degree of the freedom of the curves or the dimension of the space for hyperplane cover.

The first improvement to the aforementioned generic algorithm reduced the running time to $\mathcal{O}^*\left((k/2.2)^{dk}\right)$ for the *Line Cover* problem [17]. The best algorithm for the *Hyperplane Cover* problem, including *Line Cover*, runs in $\mathcal{O}^*\left(k^{(d-1)k}/1.3^k\right)$ time [39]. A non-parameterized solution to *Line Cover* using dynamic programming has been proposed with both time and space $\mathcal{O}^*(2^n)$ [9], which is time efficient when the number of points is $\mathcal{O}(k \log k)$. Algorithms for parabola cover and conic cover appear in [37], running in time $\mathcal{O}^*\left((k/1.38)^{(d-1)k}\right)$ and $\mathcal{O}^*\left((k/1.15)^{(d-1)k}\right)$ respectively.

Incidence Bounds. Given an arrangement of n points and m lines, an *incidence* is a point-line pair where the point lies on the line. Szemerédi and Trotter gave an asymptotic (tight) upper bound of $\mathcal{O}\left((nm)^{2/3}+n+m\right)$ on the number of incidences in their seminal paper [36]. This has inspired a long list of similar upper bounds for incidences between points and several types of varieties in different spaces, e.g. [13, 16, 32, 35].

Our Results. We give a non-parameterized algorithm solving the decision versions of both *Curve Cover* and *Hyperplane Cover* in $\mathcal{O}^*(2^n)$ time and polynomial space. Furthermore we present parameterized algorithms for *Curve Cover* and *Plane Cover* (*Hyperplane Cover* in $\mathbb{R}^3$). These solve *Curve Cover* in time $\mathcal{O}^*\left((Ck/\log k)^{(d-1)k}\right)$ and *Plane Cover* in time $\mathcal{O}^*\left((Ck^2/\log^{1/5} k)^k\right)$, both using polynomial space. The main idea is to use Szemerédi-Trotter-type incidence bounds and using the aforementioned $\mathcal{O}^*(2^n)$ algorithm as a base case. We make heavy use of (specialized) incidence bounds and our running time is very sensitive to the maximum number of possible incidences between points and curves or hyperplanes. In general, utilization of incidence bounds for constructing algorithms is rare (see e.g. [19, 20]) and to our knowledge we are the first to do so for this type of covering problem. It is generally believed that point sets that create large number of incidences must have some "algebraic sub-structure" (see e.g. [18]) but curiously, the situation is not fully understood even in two dimensions. So, it might be possible to get better specialized incidence bounds for us in the context of covering points. Thus, we hope that this work can give further motivation to study specialized incidence bounds.

[1] Throughout the paper we use the $\mathcal{O}^*(\cdot)$ notation to hide polynomial factors of a superpolynomial function.

3.2 Preliminaries

Definitions

We begin by briefly explaining the concept of fixed-parameter tractability before formally stating the *Curve Cover* and *Hyperplane Cover* problems.

Definition 3.1. A problem is said to be *fixed-parameter tractable* if there is a parameter k to an instance I, such that I can be decided by an algorithm in time $\mathcal{O}(f(k)\text{poly}(|I|))$ for some computable function f.

The function f is allowed to be any computable function, but for NP-complete problems can be expected to be at least single exponential. The name refers to the fact that these algorithms run in polynomial time when k is (bounded by) a constant. Within the scope of this paper I will typically be a set of points and k is always a solution budget: the maximum allowed size of any solution of covering objects, but not necessarily the size of the optimal such solution.

Let P be a set of n points in any dimension, and d, s be non-negative integers.

Definition 3.2. A set of algebraic curves $\mathscr{C}$ are called (d, s)-curves if (i) any pair of curves from $\mathscr{C}$ intersect in at most s points and (ii) for any d points there are at most s curves in $\mathscr{C}$ through them. The parameter d is the *degrees of freedom* and s is the *multiplicity-type*.

The set $\mathscr{C}$ could be an infinite set corresponding to a family of curves, and it is often defined implicitly. We assume two geometric predicates: First, we assume that given two curves $c_1, c_2 \in \mathscr{C}$, we can find their intersecting points in polynomial time. Second, we assume that given any set of up to $s+1$ points, in polynomial time, we can find a curve that passes through the points or decide that no such curve exists. These two predicates are satisfied in the real RAM model of computation for many families of algebraic curves and can be approximated reasonably well in practice.

We say that a curve *covers* a point, or that a point is *covered* by a curve, if the point lies on the curve. A set of curves $H \subset \mathscr{C}$ *covers* a set of points P if every point in P is covered by a curve in H, furthermore, H is a k-cover if $|H| \leq k$.

Definition 3.3 (Curve Cover Problem)**.** Given a family of (d, s)-curves $\mathscr{C}$, a set of points P, and an integer k, does there exist a subset of $\mathscr{C}$ that is a k-cover of P?

Now let P be a set of points in $\mathbb{R}^d$. A hyperplane covers a point if the point lies on the hyperplane. A set H of hyperplanes covers a set of points if every point is covered by some hyperplane; H is a k-cover if $|H| \leq k$. In $\mathbb{R}^d$, a j-flat is a j-dimensional affine subset of the space, e.g., 0-flats are points, 1-flats are lines and $(d-1)$-flats are called hyperplanes.

Definition 3.4 (Hyperplane Cover Problem)**.** Given an integer k and a set P of points in $\mathbb{R}^d$, does there exist a set of hyperplanes that is a k-cover of P?

For $d = 3$ we call the problem *Plane Cover*. To make our parameterized *Plane Cover* algorithm work, we need to introduce a third generalization: a version of *Hyperplane Cover* where the input contains any type of flats. A hyperplane covers a j-flat for $j \leq d-2$ if the flat lies on the hyperplane; further notation follows naturally from the above.

Definition 3.5 (Any-flat Hyperplane Cover Problem)**.** For $k \in \mathbb{N}$ and a tuple $P = \langle P_0, \ldots, P_{d-2} \rangle$, where P_i is a set of i-flats in $\mathbb{R}^d$, does there exist a set of hyperplanes that is a k-cover of P?

We stress that our non-parameterized algorithm in Section 3.3 solves *Any-flat Hyperplane Cover* while the parameterized algorithm in Section 3.5 solves *Plane Cover*. *Line Cover* is a special case of both *Curve Cover* and *Hyperplane Cover*. Since *Line Cover* is known to be both NP-hard [30] and APX-hard [26], the same applies to its three generalizations as well.

Kernels

Central to parameterized complexity theory is the concept of *polynomial kernels*. A parameterized problem has a polynomial kernel if an instance $\langle P, k\rangle$ in polynomial time can be reduced to an instance $\langle P', k'\rangle$ where $|P'|$ and k' are bounded by polynomial functions of k and $\langle P, k\rangle$ is a yes-instance if and only if $\langle P', k'\rangle$ is a yes-instance. Problems with polynomial kernels are immediately fixed-parameter tractable; simply run a brute force algorithm on the reduced instance.

Lemma 3.6. *For a family $\mathscr{C}$ of (d, s)-curves, Curve Cover has a size sk^2 kernel where no curve in $\mathscr{C}$ covers more than sk points.*

Proof. Suppose some curve $c \in \mathscr{C}$ covers at least $sk + 1$ points in P. These points cannot be covered by k other curves $c_1, \ldots, c_k$ as the pairwise intersection of c_i with c contains at most s points. Therefore every k-cover must include c; remove the points that it covers and decrement k. We can repeat that for every curve that covers $sk + 1$ points, until every curve in $\mathscr{C}$ covers at most sk of the remaining points. Thus, if the number of remaining points is more than sk^2, the instance has no k-cover and can be immediately rejected. Otherwise, we are left with an instance of sk^2 points. □

For *Any-flat Hyperplane Cover* a size k^d kernel is presented in [27]. It uses a *grouping* operation, removing points and replacing them with higher dimension flats, which is not acceptable for a *Hyperplane Cover* input. We present an alternative, slightly weaker hyperplane kernel containing only points; in $\mathbb{R}^3$ it contains at most $k^3 + k^2$ points.

Lemma 3.7. *Hyperplane Cover in $\mathbb{R}^d, d \geq 2$, has a size $k^2(\sum_{i=0}^{d-2} k^i) = \mathcal{O}\left(k^d\right)$ kernel where for $j \leq d - 2$ any j-flat covers at most $\sum_{i=0}^{j} k^i = \mathcal{O}(k^j)$ points and any hyperplane covers at most $k \sum_{i=0}^{d-2} = \mathcal{O}\left(k^{d-2}\right)$ points.*

Proof. This kernel boils down to creating a maximum intersection $s = \sum_{i=0}^{d-2} k^i$ between two hyperplanes. Then we have a kernel of size sk^2 where no hyperplane covers more than sk points, in exactly the same way as Lemma 3.6. For a point set P in $\mathbb{R}^d$, call a t-flat *heavy* if it covers at least $\sum_{i=0}^{t} k^t$ points in P. P is t-ready if every heavy t-flat covers exactly $\sum_{i=0}^{t} k^t$ points. When P is $(d-2)$-ready, we have the desired intersection bound s and are done. Clearly any point set is 0-ready, so we only need to show how to modify a t-ready point set into one that is equivalent and $(t+1)$-ready.

Let P be t-ready and suppose it contains a heavy $(t+1)$-flat f. Since by assumption two $(t+1)$-flats intersect in at most $\sum_{i=0}^{t} k_i$ points, any hyperplane cover of P must have a hyperplane covering f. This property is maintained by removing arbitrary points R on f so that f covers exactly $\sum_{i=0}^{t+1} k_i$ points. Consider that some of the points R were on another heavy flat f_1, which as a consequence is no longer heavy. This could mean that $P \setminus R$ has a cover where P did not, and must be rectified. We do this by re-adding enough points to f_1 so that it too covers exactly $\sum_{i=0}^{t+1} k_i$. Put these new points in general position on f_1 to avoid creating new heavy $(t+1)$-flats or increasing the number of points on any heavy t-flat. Once all heavy $(t+1)$-flats are reduced to $\sum_{i=0}^{t+1} k_i$ points in this manner, the new point set is $(t+1)$-ready and a yes-instance if and only if P is. □

Our algorithms will use both properties of the kernels. Kratsch et al. [25] showed that these kernels are essentially tight under standard assumptions in computational complexity.

Theorem 3.8 (Kratsch et al. [25]). *Line Cover has no kernel of size $\mathcal{O}(k^{2-\varepsilon})$ unless coNP $\subseteq$ NP/poly.*

Incidence bounds

Consider the *Line Cover* problem. Obviously, if the input of n points are in general position, then we need $n/2$ lines to cover them. Thus, if $k \ll \frac{n}{2}$, we expect the points to contain "some structure" if they are to be covered by k lines. Such "structures" are very relevant to the study of incidences. For a set P of points and a set L of lines, the classical Szemerédi-Trotter [36] theorem gives an upper bound on the number of point-line incidences, $I(L, P)$, in $\mathbb{R}^2$.

Theorem 3.9 (Szemerédi and Trotter [36]). *For a set P of n points and a set L of m lines in the plane, let $I(L, P) = |\{(p, \ell) \mid p \in P \cap \ell, \ell \in L\}|$. Then $I(L, P) = \mathcal{O}\Big((nm)^{2/3} + n + m\Big)$.*

The linear terms in the theorem arise from the cases when there are very few lines compared to points (or vice versa). In the setting of *Line Cover* these cases are not interesting since they are easy to solve. The remaining term is therefore the interesting one. Since it it large, it implies there are many ways of placing a line such that it covers many points; this demonstrates the importance of incidence bounds for covering problems. We introduce specific incidence bounds for curves and hyperplanes in their relevant sections.

3.3 Inclusion-exclusion algorithm

This section outlines an algorithm Inclusion-Exclusion that for both problems decides the size of the minimum cover, or the existence of a k-cover, of a point set P in $\mathcal{O}^*(2^n)$ time and polynomial space. Our algorithm improves over the one from [9] for *Line Cover* which finds the cardinality of the smallest cover of P with the same time bound but exponential space. The technique is an adaptation of the one presented in [7]; their paper immediately gives either $\mathcal{O}^*(3^n)$-time polynomial-space or $\mathcal{O}^*(2^n)$-time $\mathcal{O}^*(2^n)$-space algorithms for our problems. We give full details of the technique for completeness; to do so, we require the intersection version of the inclusion-exclusion principle.

Theorem 3.10 (Folklore). *Let $A_1, \ldots, A_n$ be a number of subsets of a universe $\mathcal{U}$. Using the notation that $\overline{A} = \mathcal{U} \setminus A$ and $\bigcap_{i \in \emptyset} \overline{A_i} = \mathcal{U}$, we have:*

$$\left| \bigcap_{i \in \{1,\ldots,n\}} A_i \right| = \sum_{X \subseteq \{1,\ldots,n\}} (-1)^{|X|} \left| \bigcap_{i \in X} \overline{A_i} \right|.$$

Curve Cover

Let P be the input set of points and $\mathscr{C}$ be the family of (d, s)-curves under consideration. Although we are creating a non-parameterized algorithm, we nevertheless assume that we have access to the solution parameter k. This assumption will be removed later. We say a set Q is a *coverable set in P* (or is *coverable* in P) if $Q \subseteq P$ and Q has a 1-cover.

Let a *tuple* (in P) be a k-tuple $\langle Q_1, \ldots, Q_k \rangle$ such that $\forall i : Q_i$ is coverable in P. Note that there is no restriction on pairwise intersection between two coverable sets in a tuple. Define $\mathcal{U}$ as the set of all tuples. For $p \in P$, let $A_p = \{\langle Q_1, \ldots, Q_k \rangle \mid p \in \bigcup_i Q_i\} \subseteq \mathcal{U}$ be the set of all tuples where at least one coverable set contains p.

Lemma 3.11. *P has a k-cover if and only if $\left|\bigcap_{p\in P} A_p\right| \geq 1$.*

Proof. Take a tuple in $\bigcap_{p\in P} A_p$. For each coverable set Q in the tuple, place a curve that covers Q. Since the tuple was in the intersection, every point is in some coverable set, so every point is covered by a placed curve. Hence we have a k-cover.

Take a k-cover $\mathcal{C}$ and from each curve $c \in \mathcal{C}$ construct a coverable set of the points covered by c. Form a tuple out of these sets and observe that the tuple is in the intersection $\bigcap_{p\in P} A_p$, hence its cardinality is at least 1. □

Note that several tuples may correspond to the same k-cover, so this technique cannot be used for the counting version of the problem. Theorem 3.10 and Lemma 3.11 reduce the problem of deciding the existence of k-covers to computing a quantity $\left|\bigcap_{i\in X} \overline{A_i}\right|$. The key observation is that $\overline{A_p}$ is the set of tuples where no coverable set contains p and $\bigcap_{i\in X} \overline{A_i}$ is the set of tuples that contain no point in X, i.e. the set of tuples in $P \setminus X$. The remainder of this section shows how to compute the size of this set in polynomial time. Let $c(X) = |\{Q \mid Q \subseteq X, Q \text{ is coverable in } X\}|$ be the number of coverable sets in a point set X. A tuple in $P \setminus X$ is k coverable sets drawn from a size $c(P \setminus X)$ pool (with replacement), hence there are $c(P \setminus X)^k$ such tuples. To compute $c(X)$ we introduce the notion of *representatives*. Let π be an arbitrary ordering of P. The representative $R = \{r_1, \ldots, r_i\}$ of a coverable set Q is the $\min(|Q|, s+1)$ first points in Q as determined by the order π. Note that for any coverable set Q, it holds that $R \subseteq Q$. Let $q(X, \pi, R)$ be the number of coverable sets that have the representative R.

Lemma 3.12. *$q(X, \pi, R)$ can be computed in $\mathcal{O}(|X|)$ time and $\mathcal{O}(\log |X|)$ space.*

Proof. If R is not a valid representative, $q(X, \pi, R) = 0$. If $|R| \leq s$, $q(X, \pi, R) = 1$. If $|R| = s+1$, let U be the union of every coverable set with representative R, and $X' = U \setminus R$. The number of subsets of X' is the number of coverable sets with representative R, i.e. $q(X, \pi, R) = 2^{|X'|}$. For any $p \in P$ with $\pi(p) > \pi(r_i)$, $p \in X'$ if and only if there is a curve $c \in \mathscr{C}$ such that c covers $\{r_1, \ldots, r_i, p\}$. Since $i \leq s+1$ the time complexity is $\mathcal{O}(|X|)$. The space complexity is logarithmic since we need only maintain $|X'|$ rather than X'. □

Lemma 3.13. *$c(X)$ can be computed in $\mathcal{O}\big(|X|^{s+2}\big)$ time and $\mathcal{O}(|X|)$ space.*

Proof. Fix an ordering π. As every coverable set in X has exactly one representative under π, we get that $c(X) = \sum_R q(X, \pi, R)$. There there are only $\mathcal{O}\Big(\binom{|X|}{s+1}\Big) = \mathcal{O}\big(|X|^{s+1}\big)$ choices of R for which $q(X, \pi, R) > 0$, and by Lemma 3.12 each term of the sum is computable in $\mathcal{O}(|X|)$ time and logarithmic space. The space complexity is therefore dominated by the space to store π which is linear. □

Theorem 3.14. *There exists a k-cover of curves from $\mathscr{C}$ for P if and only if*

$$\left|\bigcap_{p\in P} A_p\right| = \sum_{X\subseteq P} (-1)^{|X|} \left|\bigcap_{p\in X} \overline{A_p}\right| = \sum_{X\subseteq P} (-1)^{|X|} c(P \setminus X)^k \geq 1.$$

This comparison can be performed in $\mathcal{O}\big(2^n n^{s+2}\big)$ time and $\mathcal{O}(nk)$ bits of space.

Proof. Since $c(P \setminus X)^k \leq 2^{nk}$ for any X it can be stored in nk bits. The absolute value of the partial sum can be kept smaller than 2^{nk} by choosing an appropriate next X. The rest follows from Theorem 3.10, Lemma 3.11 and Lemma 3.13. □

Finally, we remove the assumption that we have the parameter k. Any input requires at most n curves. Since k is only used to compute $c(X)^k$ we can try $k = 1, 2, \ldots, n$ and return the first k with a positive sum. This increases the time by an $\mathcal{O}(n)$ factor. Alternatively, we can run n simultaneous sums, since the parameter k is only accessed when computing $c(X)^k$. This increases the space by factor $\mathcal{O}(n)$ and the time by a lower-order additive term.

Any-flat Hyperplane Cover

Here we treat all flats in the instance $\langle P_0, \ldots, P_{d-2} \rangle$ as atomic objects and P as a union $\bigcup_{i=0}^{d-2} P_i$. This algorithm is very similar to that of Section 3.3, so we only describe their differences. A set of flats $Q \subseteq P$ is a coverable set in P if there exists a hyperplane that covers every $p \in Q$. The representative of $\emptyset$ is $\emptyset$, and the representative of a non-empty coverable set Q is a set $R = \{r_1, \ldots, r_i\}$. Let r_1 be the first flat in Q and for $j \geq 2$, r_j is defined if the affine hull of $\{r_1, \ldots, r_{j-1}\}$ has lower dimension than the affine hull of Q. If so, let r_j be the first flat in Q that is not covered by the affine hull of $\{r_1, \ldots, r_{j-1}\}$.

Lemma 3.15. *$q(X, \pi, R)$ can be computed in $\mathcal{O}(|X|)$ time and $\mathcal{O}(\log |X|)$ space.*

Proof. If R is not a valid representative, $q(X, \pi, R) = 0$. Otherwise, let U be the union all coverable sets with the representative R, and $X' = U \setminus S$. For every $p \in X \setminus R$, let j be the highest index such that $\pi(r_j) < \pi(p)$. Then $p \in X'$ if and only if p is on the affine hull of $\{r_1, \ldots, r_j\}$. □

There are $\mathcal{O}(\binom{n}{d})$ representatives R with $q(X, \pi, R) > 0$ so the following two results hold; their proofs are analogous to Lemma 3.13 and Theorem 3.14.

Lemma 3.16. *$c(X)$ may be computed in $\mathcal{O}\left(|X|^{d+1}\right)$ time and $\mathcal{O}(|X|)$ space.*

Theorem 3.17. *There exists a hyperplane k-cover for P if and only if*

$$\left|\bigcap_{p \in P} A_p\right| = \sum_{X \subseteq P} (-1)^{|X|} \left|\bigcap_{p \in X} \overline{A_p}\right| = \sum_{X \subseteq P} (-1)^{|X|} c(P \setminus X)^k \geq 1.$$

This comparison may be performed in $\mathcal{O}\left(2^n n^{d+1}\right)$ time and $\mathcal{O}(nk)$ bits of space.

3.4 Curve Cover

Recall that we are considering (d, s)-curves, where d and s are constants. Since we have a kernel of up to sk^2 points, Inclusion-Exclusion used on its own runs in time $\mathcal{O}^*\left(2^{sk^2}\right)$ which is too slow to give an improvement. We improve this by first using a technique that reduces the number of points in the input, and then using Inclusion-Exclusion. To describe this technique and the intuition behind it, we first provide a framework based on the following theorem by Pach and Sharir.

Theorem 3.18 (Pach and Sharir [32])**.** *Let P be a set of n points and L a set of m (d, s)-curves in the plane. The number of point-curve incidences between P and L is*

$$I(P, L) = \mathcal{O}\left(n^{d/(2d-1)} m^{(2d-2)/(2d-1)} + n + m\right)$$

Note that the above holds for curves in arbitrary dimension. This can be seen by projecting the points and curves onto a random plane, which will keep the projection of distinct points, and prevent the curves from projecting to overlapping curves.

Definition 3.19. Let a *candidate* be any curve in $\mathscr{C}$ that covers at least 1 point in P. Define its *richness* with respect to P as the number of points it covers. A candidate is γ-rich if its richness is at least γ, and γ-poor if its richness is at most γ.

Recall that from the kernelization in Lemma 3.6, it follows that every candidate is sk-poor. The following gives a bound on the number of γ-rich candidates.

Lemma 3.20. *Let P be a set of n points in some finite dimension space $\mathbb{R}^x$. The number of γ-rich candidates in P is $\mathcal{O}\Big(\frac{n^d}{\gamma^{2d-1}} + \frac{n}{\gamma}\Big)$*

Proof. If m curves pass through γ or more points, this generates at least $m\gamma$ incidences. By Theorem 3.18 we get

$$m\gamma = \mathcal{O}\Big(n^{d/(2d-1)}m^{(2d-2)/(2d-1)} + n + m\Big).$$

We deal with the three terms in the $\mathcal{O}(\cdot)$ separately. If $m\gamma = \mathcal{O}\Big(n^{d/(2d-1)}m^{(2d-2)/(2d-1)}\Big)$, the expression simplifies to $m = \mathcal{O}\Big(n^d\gamma^{-(2d-2)}\Big)$. If $m\gamma = \mathcal{O}(n)$ we have that $m = \mathcal{O}(n/\gamma)$. If $m\gamma = \mathcal{O}(m)$ then γ is a constant. Since at most s curves pass through the same d points, m is bounded by the total number of distinct curves $s\binom{n}{d} = \mathcal{O}\Big(n^d\Big)$. Therefore, this case is covered by the first term, giving the total bound $m = \mathcal{O}\Big(\frac{n^d}{\gamma^{2d-1}} + \frac{n}{\gamma}\Big)$. □

Intuition for algorithm. We exploit the following observation: given a k-cover $\mathcal{C}$, some curves in $\mathcal{C}$ might be significantly richer than others. The main idea of our technique is to try to select (i.e. branch on) these rich curves first. Since they cover "many" points, removing these decreases the ratio $|P|/k$ and calling Inclusion-Exclusion eventually becomes viable. The idea to branch on rich curves first has another important consequence. Suppose we know that no candidate in $\mathscr{C}$ covers more than γ points in P. This immediately implies that if there are strictly more than $k\gamma$ points in P, it is impossible to cover P. Therefore we have $|P|/k \leq \gamma$. Now look at the set of $\frac{\gamma}{2}$-rich candidates and decide for each whether to include it in the cover or not. By the earlier observation, including such a candidate is good for reducing the ratio $|P|/k$. But excluding such a candidate has essentially the same effect, because that candidate will not be considered again (remove it from $\mathcal{C}$). Any remaining candidates in $\mathcal{C}$ now cover at most $\frac{\gamma}{2}$ points; we must have $|P|/k \leq \frac{\gamma}{2}$ (or the instance is not solvable) and have strengthened the bound on the ratio. Regardless of which choice we make, we make progress towards being able to call the base case.

This strategy also makes sense from a combinatorial point of view, because from Lemma 3.20 it follows that the search space is small for rich curves. Switching to Inclusion-Exclusion early enough lets us bypass the potentially very large search space of poor candidates.

The Algorithm. Let r be a parameter. The exact value is set in the proof of Theorem 3.25, for now it is enough that $r = \Theta(\log k)$. For a budget k let $\langle k_1, \ldots, k_r\rangle$ with $\sum_j k_j = k$ be a *budget partition*. We describe a main recursive algorithm CC-Recursive (see the appendix for pseudocode) that takes 4 arguments: the point set P, the class of curves $\mathscr{C}$, a budget partition $\langle k_1, \ldots, k_r\rangle$, and a recursion level i. For convenience we define $\gamma_i = sk/2^i$. A simple top-level procedure CurveCover tries all budget partitions and calls the recursive algorithm with that partition at recursion level 1.

At every recursion depth i, let $K_i = \sum_{j=i}^{r} k_j$ be the remaining budget and P_i the remaining point set. That means earlier levels have created a partial solution $\mathcal{C}_{i-1}$ of $k - K_i$ curves covering the points $P \setminus P_i$. The recursive algorithm will try to cover the remaining points using γ_{i-1}-poor curves. Specifically, at depth i let S be the set of candidates from $\mathscr{C}$ that are γ_i-rich and

γ_{i-1}-poor. Since from depth i and onward it has a remaining budget of K_i and cannot pick candidates that are $(\gamma_{i-1}+1)$-rich, the algorithm rejects if strictly more than $K_i\gamma_{i-1}$ remain. If fewer than $\frac{(d-1)}{2}K_i \log k$ points remain, the sub problem is solved with inclusion-exclusion.

If neither a reject (due to too many points) or a base-case call to inclusion-exclusion has occurred, the algorithm will branch. It does so in $\binom{|S|}{k_i}$ ways by simply trying all ways of choosing k_i candidates from S. For each such choice, all points in P covered by the chosen candidates are removed and the algorithm recurses to depth $i+1$. If all those branches fail, the instance is rejected.

Analysis

Lemma 3.21. *Algorithm* CurveCover *decides whether P has a k-cover of curves from $\mathscr{C}$.*

Proof. Regard CurveCover as being non-deterministic. Suppose P has a k-cover $\mathcal{C}$. The proof is by induction on the recursion. Assume as the induction hypothesis that the current partial solution $\mathcal{C}_{i-1}$ is a subset of $\mathcal{C}$ and that $\mathcal{C}$ contains no curves that are $(\gamma_{i-1}+1)$-rich when restricted to P_i. The assumption is trivially true for $i=1$ as $\mathcal{C}_0=\emptyset$.

By the induction hypothesis, $\mathcal{C}\setminus\mathcal{C}_{i-1}$ is a K_i-cover for P_i using only γ_{i-1}-poor curves. Therefore it holds that $|P_i| \leq \gamma_{i-1}K_i$, and thus the algorithm does not reject incorrectly. Furthermore, if Inclusion-Exclusion is called it accepts since we are in the case that a solution exists.

Otherwise, let $D \subseteq \mathcal{C}\setminus\mathcal{C}_{i-1}$ be the curves that are γ_i-rich when restricted to P_i. The algorithm non-deterministically picks D from the set of candidates S and constructs $\mathcal{C}_i = \mathcal{C}_{i-1}\cup D$. This leaves $\mathcal{C}_i$ to be a subset of $\mathcal{C}$. Additionally, $\mathcal{C}_i$ contains all γ_i-rich curves in $\mathcal{C}$ restricted to P_i and hence to $P_{i+1}\subseteq P_i$, upholding the induction hypothesis.

Suppose the algorithm accepts the instance $\langle P,k\rangle$. It can only accept if some call to Inclusion-Exclusion accepts. Let $\mathcal{C}_r$ be the set of curves selected by the recursive part such that Inclusion-Exclusion accepted the instance $\langle P\setminus\mathcal{C}_r, k-|\mathcal{C}_r|\rangle$. Let $\mathcal{C}_{ie}$ be any $(k-|\mathcal{C}_r|)$-cover of $P\setminus\mathcal{C}_r$. Then $\mathcal{C}_r\cup\mathcal{C}_{ie}$ is a k-cover of P. □

By the nature of the inclusion-exclusion algorithm, CurveCover detects the existence of a k-cover rather than producing one. But since CC-Recursive produces a partial cover during its execution, it is straight-forward to extend that into a full k-cover by using Inclusion-Exclusion as an oracle.

Running time. To analyze the running time of the algorithm we see the execution of CC-Recursive as a search tree $\mathcal{T}$. Each leaf of the tree is either an immediate reject or a call to Inclusion-Exclusion. Since the latter is obviously most costly to run, we must assume for a worst case analysis that every leaf node calls the base case algorithm. The running time is the number of leaf nodes in the search tree times the running time of Inclusion-Exclusion. Since the algorithm performs exponential work in these leaf nodes but not in inner nodes, it is insufficient to reason about the *size* of the tree. Therefore we will speak of the "running time of a subtree", which simply means the running time of the recursive call that corresponds to the root of that subtree. We show that in the worst case, $\mathcal{T}$ is a complete tree $\mathcal{T}_1$ of depth r. That is, $\mathcal{T}_1$ has no leaf nodes at depths less than r.

Let $\mathcal{T}_j$ be a complete subtree of $\mathcal{T}_1$ rooted at depth j. To prove that $\mathcal{T}_1$ is the worst case for $\mathcal{T}$ we prove two things. First we first prove an upper bound on the running time for arbitrary $\mathcal{T}_j$. Then we prove that the running time of $\mathcal{T}_1$ can only improve if an arbitrary subtree is replaced by a leaf (i.e. a call to Inclusion-Exclusion). The most involved part is proving an upper bound on the number of leaves of $\mathcal{T}_j$.

Lemma 3.22. *Let L be the number of leaves in $\mathcal{T}_j$. Then for some constant $c_2 = c_2(d, s)$, L is bounded by*

$$L \leq \left(\frac{c_2 k^d}{(k - k_r) \log^{d-1} k} \right)^{K_j - k_r}$$

The proof is long and tedious and we leave it for the appendix. To give an idea of how Lemma 3.22 is proved, we sketch a simplified worst case analysis for *Line Cover*. The analysis can be generalized to *Curve Cover* and gives (up to a constant in the base of the exponent) the same running time as the real worst case.

Analysis sketch. The branching of $\mathcal{T}_1$ at recursion level i depends on the budget k_i that is being used. That means that the structure of the whole tree depends on the complete budget partition. From Lemma 3.20 it follows that the lower the richness the more candidates there are. Since the richness halves after every recursive call, one could conjecture that the worst case budget partition would put as much budget in the end. It could e.g. look like $\langle 0, 0, \ldots, 0, k_{r-1}, k_r \rangle$, where $k - k_r = k_{r-1} > k_r$. That is, only in the penultimate and last recursion level is there any budget to spend. At the deepest level of recursion, the richness considered is strictly less than $\frac{\log k}{2}$ (because with this richness the base case algorithm is efficient). Therefore, at the penultimate recursion level the richness is $\log k$. At this level there are $k \log k$ points left and we can apply Lemma 3.20 to bound the number of $\log k$ rich lines. This yields a bound of $\frac{k^2}{\log k}$ on the number of candidates. From these we pick $k - k_r$ lines, giving a branching of roughly $\left(\frac{k^2}{(k-k_r)\log k}\right)^{k-k_r}$ (where roughly means up to a constant in the base of the exponent).

It turns out that the worst case budget partition is in fact $\langle k_0 2^1, k_0 2^2, ..., k_0 2^{r-1}, k_r \rangle$ for some k_0. However, to understand where the division by $\log^{d-1} k$ comes from in the expression of Lemma 3.22, it is sufficient to understand the above analysis sketch. With Lemma 3.22 in place, we can prove the following bound on the running time of $\mathcal{T}_j$.

Lemma 3.23. *The time complexity of a complete subtree $\mathcal{T}_j$ is $\mathcal{O}^*\Big((c_4 k / \log k)^{(d-1)K_j}\Big)$, where $c_4 = c_4(d, s)$ is a constant that depends on the family $\mathscr{C}$.*

Proof. By Lemma 3.22, the number of leaves in $\mathcal{T}_j$ is $L \leq \left(\frac{c_2 k^d}{(k-k_r)\log^{d-1} k}\right)^{K_j - k_r}$. Observe that at depth r, Inclusion-Exclusion runs in time $\mathcal{O}^*\Big(2^{\frac{d-1}{2} k_r \log k}\Big) = \mathcal{O}^*\Big((k^{1/2})^{(d-1)k_r}\Big)$. Since an inner node performs polynomial time work and the leaves perform exponential time work, this immediately implies that the running time for $\mathcal{T}_j$ is

$$\mathcal{O}^*\left(\left(\frac{c_2 k^d}{(k - k_r) \log^{d-1} k} \right)^{K_j - k_r} \cdot (k^{1/2})^{(d-1)k_r} \right).$$

Suppose that $k - k_r \geq c_3 k$ for some constant $c_3 > 0$, then the running time solves to:

$$\mathcal{O}^*\left(\left(\frac{c_2 k^{d-1}}{c_3 \log^{d-1} k} \right)^{K_j - k_r} \cdot (k^{1/2})^{(d-1)k_r} \right) = \mathcal{O}^*\left(\left(\frac{c_4 k}{\log k} \right)^{(d-1)K_j} \right)$$

where $c_4 = (c_2/c_3)^{1/(d-1)}$.

When $k - k_r$ is less than a constant fraction of k, that is $k - k_r = o(k)$, it holds that $K_j - k_r = o(K_j)$ since $k \geq K_j \geq k_r$.

$$\mathcal{O}^*\left(\left(\frac{c_2 k^d}{(k - k_r) \log^{d-1} k} \right)^{o(K_j)} \cdot \left(k^{1/2}\right)^{(d-1)(K_j - o(K_j))} \right) = \mathcal{O}^*\Big(2^{o(dK_j \log k) + \frac{d-1}{2}(K_j \log k - o(k \log k))}\Big)$$

With some simple algebra one gets that the exponent is bounded by $(d-1)K_j(\log k - \log\log k)$, giving the desired time bound $\mathcal{O}^*\Big(2^{(d-1)K_j(\log k-\log\log k)}\Big) = \mathcal{O}^*\Big((k/\log k)^{(d-1)K_j}\Big)$. □

Lemma 3.24. *Let L_j be a depth $j < r$ leaf of $\mathcal{T}$ that calls* Inclusion-Exclusion. *Then the running time of $\mathcal{T}_j$ dominates that of L_j.*

Proof. By Lemma 3.23, the time complexity of $\mathcal{T}_j$ is $\mathcal{O}^*\Big((c_4k/\log k)^{(d-1)K_j}\Big)$. At depth j the algorithm has K_j remaining budget to spend. Since the algorithm called Inclusion-Exclusion at this depth, at most $\frac{d-1}{2}K_j \log k$ points remained and the call takes $\mathcal{O}^*\Big(2^{\frac{d-1}{2}K_j\log k}\Big) = \mathcal{O}^*\Big((k^{1/2})^{(d-1)K_j}\Big)$ time, which is bounded by that for $\mathcal{T}_j$. □

Theorem 3.25. CurveCover *decides Curve Cover in time $\mathcal{O}^*\Big((Ck/\log k)^{(d-1)k}\Big)$ where $C = C(d,s)$ is a constant that depends on the family $\mathscr{C}$.*

Proof. Fix a budget partition $\langle k_1, \ldots, k_r\rangle$. By Lemma 3.24, calling Inclusion-Exclusion at a depth $j < r$ does not increase the running time of the algorithm. Therefore the time complexity of CC-Recursive is $\mathcal{O}^*\Big((c_4k/\log k)^{(d-1)K_1}\Big) = (c_4k/\log k)^{(d-1)k}$.

CurveCover runs CC-Recursive over all possible budget partitions, of which by the "stars and bars" theorem are only $\binom{k+r-1}{k}$, a quasi-polynomial in k. Therefore by letting $C = c_4 + \varepsilon$ for any $\varepsilon > 0$, the time complexity of CurveCover is $\mathcal{O}^*\Big((Ck/\log k)^{(d-1)k}\Big)$. □

Lemma 3.26. *The polynomial time dependency of* CurveCover *is $\mathcal{O}\big((k\log k)^{2+s}\big)$ and its space complexity is $\mathcal{O}\Big(k^4\log^2 k\Big)$ bits.*

Proof. The height of the tree is $r = \mathcal{O}(\log k)$. Inner nodes have polynomial time and space which is strictly dominated by the exponential time and polynomial space of the leaves. Hence the polynomial time dependency of CurveCover is exactly the polynomial time dependency of the leaves. Inclusion-Exclusion runs in $\mathcal{O}(n^{s+2}2^n)$ and $n = \mathcal{O}(k\log k)$ points remain when it is called; the polynomial dependency is $\mathcal{O}\big((k\log k)^{s+2}\big)$.

Inclusion-Exclusion requires only $\mathcal{O}(nk) = \mathcal{O}(k^2\log k)$ bits of storage, while an inner node stores its set of candidates S. A trivial bound on the size of any S is $\binom{sk^2}{2}$ elements, which can be stored in $\mathcal{O}(k^4\log k)$ bits. Since $r = \Theta(\log k)$, we use no more than $\mathcal{O}\Big(k^4\log^2 k\Big)$ bits to store them. □

3.5 Hyperplane Cover

One generalization of *Line Cover* was discussed in the previous section. In this section we discuss its other generalization *Hyperplane Cover*, and give an algorithm for the three dimensional case. We would like to follow the same basic attack plan of using incidence bounds but here we face significant challenges and we need non-trivial changes in our approach. One major challenge is the nature of incidences in higher dimensions. For example, the asymptotically maximum number of incidences between a set of points and hyperplanes in d-dimensions is obtained by placing half of the points on one two-dimensional plane (see [2, 11]) which clearly makes it an easy instance for our algorithm (due to kernelization). Thus, in essence, we need to use specialized incidence bounds that disallow such configurations of points; unfortunately, such bounds are more difficult to prove than ordinary incidence bounds (and as it turns out, also more difficult to use).

Point-Hyperplane incidence bounds in higher dimensions

The most general bound for point-hyperplane incidences from [2, 12] yields a bound of $\Theta\left(\frac{n^d}{\gamma^3} + \frac{n^{d-1}}{\gamma}\right)$ on the number of γ-rich hyperplanes in d dimensions similar to Lemma 3.20 (where the left term is again the significant one). Our method requires that the exponent is greater in the denominator than in the numerator, so this bound is not usable beyond $\mathbb{R}^2$. As stated before, the constructions that make the upper bound tight are easy cases for our algorithm; they contain very low dimensional flats that have many points on them. A specialized bound appears in [13], where the authors study the number of incidences between points and hyperplanes with a certain *saturation*.

Definition 3.27. Consider a point set P and a hyperplane H in $\mathbb{R}^d$. We say that H is σ-saturated, $\sigma > 0$, if $H \cap P$ spans at least $\sigma \cdot |H \cap P|^{d-1}$ distinct $(d-2)$-flats of F.

For example in three dimensions, a $(1-\frac{1}{n})$-saturated plane contains no three collinear points. The main theorem of [13] can be stated as follows.

Theorem 3.28 (Elekes and Tóth [13])**.** *Let $d \geq 2$ be the dimension and $\sigma > 0$ a real number. There is a constant $C_1(d, \sigma)$ with the following property. For every set P of n points in $\mathbb{R}^d$, the number of γ-rich σ-saturated hyperplanes is at most:*

$$\mathcal{O}\left(C_1(d,\sigma)\left(\frac{n^d}{\gamma^{d+1}} + \frac{n^{d-1}}{\gamma^{d-1}}\right)\right).$$

The interesting term in this bound has a greater exponent in the denominator, as required. An issue is that it is not easy to verify if a hyperplane is σ-saturated. In the same paper as Theorem 3.28, the authors give another bound based on a more manageable property called *degeneracy*.

Definition 3.29. Given a point set P and a hyperplane H in $\mathbb{R}^d$, we say that H is δ-degenerate, $0 < \delta \leq 1$, if $H \cap P$ is non-empty and at most $\delta \cdot |H \cap P|$ points of $H \cap P$ lie in any $(d-2)$-flat.

For example in $\mathbb{R}^3$, any 1-degenerate plane might have all its points lying on a single line, and a plane with degeneracy strictly less than 1 must have at least 3 points not on the same line. As such it is an easy property to test.

Theorem 3.30 (Elekes and Tóth [13])**.** *For any set of n points in $\mathbb{R}^3$, the number of γ-rich δ-degenerate planes is at most*

$$\mathcal{O}\left(\frac{1}{(1-\delta)^4}\left(\frac{n^3}{\gamma^4} + \frac{n^2}{\gamma^2}\right)\right).$$

This bound is usable and relies on an easily-tested property, but unfortunately only applies to the $\mathbb{R}^3$ setting.

Algorithm for *Plane Cover*

In this section we present our algorithm PC-Recursive that solves *Plane Cover* using the bound from Theorem 3.30. This algorithm is similar to the algorithm for *Curve Cover*, and it is assumed that the reader is sufficiently familiar with CC-Recursive before reading this section.

Recall that by Lemma 3.7, *Plane Cover* has a kernel of size $k^3 + k^2$ where no plane contains more than $k(k+1) \leq 2k^2$ points and no two planes pairwise intersect in more than $k+1$ points.

For convenience we define $\gamma_0 = k^2 + k$ and $\gamma_i = k^2/2^i$ for $i > 0$. We inherit the basic structure of the CC-Recursive algorithm, such that every recursion level considers γ_i-rich-γ_{i-1}-poor candidates. Additionally, any candidate considered must be *not-too-degenerate*:

Definition 3.31. Let $\delta_i = 1 - \gamma_i^{-1/5}$. A γ_i-rich-γ_{i-1}-poor plane is called *not-too-degenerate* if it is δ_i-degenerate, and *too-degenerate* otherwise.

It is of no consequence that the definition does not cover all candidates considered on depth 1. The main extension of PC-Recursive compared to CC-Recursive is to first use a different technique to deal with too-degenerate candidates, which then allows normal branching on the not-too-degenerate ones. The key observation is that any too-degenerate candidate has at least $\gamma_i\delta_i = \gamma_i - \gamma_i^{4/5}$ points on a line and at most $\gamma_{i-1}(1-\delta_i) = 2\gamma_i^{4/5}$ points not on it.

Suppose a k-cover contains some too-degenerate plane H. By correctly guessing its very rich line L and removing the points on the line, the algorithm makes decent progress in terms of shrinking the instance. The points on H but not L will remain in the instance even though the budget for covering them has been paid. These are called the *ghost points* of H (or of L), and L is called a *degenerate line*. The ghost points must be removed by extending the line L into a full plane. But the ghost points are few enough that the algorithm can delay this action until a later recursion level. Specifically, for a line L guessed at depth i, we extend L into a plane at the first recursion depth which considers $2\gamma_i^{4/5}$-poor candidates, i.e. the depth j such that $\gamma_{j-1} \geq 2\gamma_i^{4/5} \geq \gamma_j$.

Therefore the algorithm keeps a separate structure $\mathcal{L}$ of lines that have been guessed to be degenerate lines on some planes in the solution. Augment $\mathcal{L}$ to remember the recursion depth that a line was added to it. At any recursion depth, the algorithm will deal with old-enough lines in $\mathcal{L}$, then guess a new set of degenerate lines to add to $\mathcal{L}$ before finally branching on not-too-degenerate planes.

The algorithm Let $r = \Theta(\log k)$ as before. Let $\langle h_1, \ell_1, \ldots, h_r, \ell_r\rangle$ with $\sum_{i=1}^{r} h_i + \ell_i = k$ be a budget partition. The recursive algorithm PC-Recursive takes 4 arguments: the point set P, a set of lines $\mathcal{L}$, the budget partition, and a recursion level i. A top level algorithm PlaneCover tries all budget partitions and calls PC-Recursive accordingly.

Let the current recursion depth be i, and let $K_i = \sum_{j=i}^{r} h_j + \ell_j$ be the remaining budget. The sub-budget h_i will be spent on not-too-degenerate planes, and ℓ_i on degenerate lines. Let $\mathcal{L}$ be an augmented set of lines as described above. This means that earlier levels have already created a partial solution of $k - (K_i + |\mathcal{L}|)$ planes, and a set $\mathcal{L}$ of lines that still need to be covered by a plane. If strictly more than $(K_i + |\mathcal{L}|)\gamma_{i-1}$ points remain, the algorithm rejects. If at most $K_i \log k$ points, the algorithm switches to Inclusion-Exclusion passing on the instance $\langle P \cup \mathcal{L}, K_i + |\mathcal{L}|\rangle$.

Let $f = \left\lceil \frac{5(i-1) - 2\log k}{4} \right\rceil$. Let A be the set of all lines in $\mathcal{L}$ that were added at depth f or earlier. Remove A from $\mathcal{L}$. For each way of placing $|A|$ planes $\mathcal{H}$ such that every plane contains one line in A and at least one point in P, let $P' = P \setminus (P \cap \mathcal{H})$ be the point set not covered by these planes. For a P', let H be the set of not-too-degenerate planes and L the set of degenerate lines too-degenerate candidates.

For every P' and every way of choosing h_i planes from H and ℓ_i lines from L, branch depth $i+1$ by removing the covered points from P and adding the chosen lines of L to $\mathcal{L}$.

Analysis

Correctness. To prove that the algorithm is correct, we follow a similar strategy as for CurveCover. We build on the notion that the algorithm is building up a partial solution of

planes. Removing the points covered by the partial solution yields a "residual problem" just as in CurveCover. A partial solution is *correct* if it is a subset of some k-cover. Correctness of the algorithm follows from proving that a k-cover exists if and only if one branch maintains a correct partial solution until it reaches Inclusion-Exclusion.

The difference here is that the residual problem is an instance of *Any-flat Plane Cover* and not *Plane Cover*. Therefore, we simply consider the original problem to be an instance of *Any-flat Plane Cover*, namely $R_1 = \langle P, \emptyset \rangle$. We say that $\mathcal{C}$ covers $\langle P, \mathcal{L} \rangle$ if $\mathcal{C}$ covers both P and $\mathcal{L}$. What needs to be established is that there is a correct way to replace points with lines (Observation 3.32) and, conversely, that there is a correct way to extend a line in $\mathcal{L}_i$ (Observation 3.33). The proofs for these are elementary and we omit them. Given these two facts, we can easily show that the algorithm will call Inclusion-Exclusion on appropriate instances.

Observation 3.32. Let ℓ be a line and $\mathcal{C}$ a set of planes such that some plane $h \in \mathcal{C}$ covers ℓ. Then $\mathcal{C}$ is a cover for $\langle P, \mathcal{L} \rangle$ if and only if $\mathcal{C}$ is a cover for $\langle P \setminus \ell, \mathcal{L} \cup \{\ell\} \rangle$.

Observation 3.33. Let ℓ be a line, $\mathcal{L} \ni \ell$ be a set of lines, and $\mathcal{C}$ be a set of planes such that some $h \in \mathcal{C}$ covers ℓ but not any other line $\ell' \in \mathcal{L}$. Then $\mathcal{C}$ is a cover for $\langle P, \mathcal{L} \rangle$ if and only if $\mathcal{C} \setminus \{h\}$ is a cover of $\langle P \setminus h, \mathcal{L} \setminus \{\ell\} \rangle$.

The conditions for Observation 3.33 might seem overly restrictive. But as the following Lemma 3.34 shows, that situation arises when $\mathcal{L}$ contains only correctly guessed degenerate lines.

Lemma 3.34. *Let h be a too-degenerate plane with degenerate line ℓ such that $h \setminus \ell$ is a too-degenerate plane with degenerate line ℓ'. Then at no point during the execution of* PC-Recursive *will $\mathcal{L}$ contain ℓ and ℓ'.*

Proof. Let j_1 be the depth that ℓ was put in $\mathcal{L}$, and j_2 the depth for ℓ'. Since ℓ is a degenerate line, it has at most $2\gamma_j^{4/5}$ ghost points. It gets removed from $\mathcal{L}$ on some depth i where $j_1 \leq f = \left\lceil \frac{5(i-1)-2\log k}{4} \right\rceil$. Since ℓ' was put in $\mathcal{L}$ before ℓ was taken out we have $j_2 \leq i-1$ and $j_1 > \frac{5(j_2)-2\log k}{4}$. This implies that $\frac{k^2}{2^{j_2}} > \left(\frac{k^2}{2^{j_1}}\right)^{4/5}$ or $\gamma_{j_2} > \gamma_{j_1}^{4/5}$. Since ℓ_2 is also a degenerate line, it was on a γ_{j_2}-rich candidate. This candidate contains more points than the possible number of ghost points of ℓ, so ℓ was not a degenerate line. □

Lemma 3.35. *If $\mathcal{L}$ contains only the degenerate lines of some too-degenerate planes in a k-cover, the number of ghost points at depth i is at most $|\mathcal{L}|\gamma_{i-1}$.*

Proof. Consider a degenerate line $\ell \in \mathcal{L}$ guessed at some recursion depth $j < i$. Line ℓ was on a δ_j-degenerate plane, i.e. it covered at least $\gamma_j\delta_j$ points and has at most $2\gamma_j^{4/5}$ ghost points. Since ℓ was not removed from $\mathcal{L}$ at depth $i-1$, we get $j > f_{i-1} = \left\lceil \frac{5((i-1)-1)-2\log k}{4} \right\rceil$. Some simple algebra gives $\frac{k^2}{2^{i-2}} \geq \left(\frac{k^2}{2^j}\right)^{4/5}$, i.e. $\gamma_{i-1} \geq 2\gamma_j^{4/5}$. Hence any line in $\mathcal{L}$ has left at most γ_{i-1} ghost points in the instance, and the sum of ghost points is at most $|\mathcal{L}|\gamma_{i-1}$. □

Lemma 3.36. *Algorithm* PlaneCover *decides whether P has a k-cover of planes.*

Proof. View the algorithm as being non-deterministic. Suppose P has a k-cover. Observation 3.32, Observation 3.33 and Lemma 3.34 guarantee that there is a correct path, and Lemma 3.35 guarantees that the point set is not erroneously rejected. Therefore the algorithm will send a yes-instance to Inclusion-Exclusion and accept.

Suppose P has no k-cover. If the conditions for Observation 3.33 are not satisfied, removing ℓ from $\mathcal{L}$ and pairing it up with points but not with another $\ell' \in \mathcal{L}$ can only reduce the number of solutions. Therefore the algorithm detects no cover and rejects. □

We can now state our main theorem for Plane-Cover.

Theorem 3.37. PlaneCover *decides Plane Cover in* $\mathcal{O}\Big((Ck^2/\log^{1/5} k)^k\Big)$ *time for some constant* C.

To give an idea of how to prove the above theorem, we give a sketch of the analysis that reflects the core of the real analysis. As before, we assume a (slightly incorrect) worst case for the budget partition where all the budget is assigned to the two deepest recursion levels. This gives a bound analogous to the bound appearing in Lemma 3.22. After achieving this bound, the same arguments as for Curve-Cover can be applied to achieve the bound from Theorem 3.37.

Analysis sketch. The branching of the analysis is twofold. First there is the branching done on picking not-too-degenerate planes. Secondly, we have the branching on too-degenerate planes. This branching is actually a combination of picking the rich lines in too-degenerate planes, and the branching done by covering these lines with planes later on.

We sketch a bound here for the cases that either all the budget goes into picking not-too-degenerate planes, *or* all budget goes into picking too-degenerate planes (i.e. lines). We show that if either (**i**) $\forall i, k_i = h_i$ or (**ii**) $\forall i, \ell_i = k_i$, then the branching can be bounded by $\left(\frac{k^3}{(k-k_r)\log^{1/5} k}\right)^{k-k_r}$ (compare to Lemma 3.22). The full proof for Theorem 3.37 shows that if the budget is distributed between these cases, then taking the product of the worst case running times of both cases is roughly the same as what we present here. For both cases, we again assume a (slightly incorrect) worst case budget partition where $k_{r-1} + k_r = k$ and $k_{r-1} \geq k_r$. By the same arguments as in the analysis sketch of Section 3.4 we have the following two parameters at recursion level $r-1$; the number of of points remaining is $n = k \log k$ and the richness γ_{r-1} is $\log k$.

For (**i**) we can directly apply Theorem 3.30 as follows. For the term $\frac{1}{(1-\delta)^4}$ we can substitute δ with $1 - \gamma_{r-1}^{-1/5} = 1 - \log^{-1/5} k$ to get $\log^{4/5} k$. Plugging in all these values in Theorem 3.30 we get that the number of not-too-degenerate planes is bounded by $\frac{\log^{4/5} n^3}{\log^4 k} = \frac{k^3}{\log^{1/5} k}$. From these candidates we pick $k - k_r$ planes, giving a branching of $\binom{\frac{k^3}{\log^{1/5} k}}{k-k_r}$, which is roughly $\left(\frac{k^3}{(k-k_r)\log^{1/5} k}\right)^{k-k_r}$.

For (**ii**) we do the following. The algorithm picks γ_{i+1}-rich lines at level i, and these lines are matched with points at later level j where $\gamma_j = \gamma_i^{4/5}$. The cost for branching at level j is charged to level i, so that we can more easily analyze the total branching on lines selected at level i. With the budget partition as stated above, we can now bound the branching done at level $r-1$. By the Szemerédi-Trotter theorem, there are at most $\frac{n^2}{(\log k)^3} = \frac{k^2}{\log k}$ candidates, from which we select $k - k_r$ lines. This yields a total branching of $\binom{\frac{k^2}{\log k}}{k-k_r}$, which is roughly $\left(\frac{k^2}{(k-k_r)\log k}\right)^{k-k_r}$. We then need to mach these $k - k_r$ lines with $k \log^{4/5} k$ points, yielding a further branching of $(k \log^{4/5} k)^{k-k_r}$. Taking the product of both these branching factors gives $\left(\frac{k^2}{(k-k_r)\log k}\right)^{k-k_r} \cdot (k \log^{4/5} k)^{k-k_r} = \left(\frac{k^3}{(k-k_r)\log^{1/5} k}\right)^{k-k_r}$.

Proof of Theorem 3.37. It holds that $\sum_{i=1}^{r} h_i + \sum_{i=1}^{r} \ell_i = \sum_{i=1}^{r} k_i = k - K_r$, so for convenience we define ε such that $\varepsilon(k - K_r) = \sum_{i=1}^{r} h_i$.

By the bound we have that the number of not-too-degenerate planes at level i is:

$$\left(\frac{1}{1-\delta_i}\right)^4 \frac{(k\gamma_{i-1})^3}{(\gamma_i)^4} = \mathcal{O}\left(\frac{k^3}{\gamma_i^{1/5}}\right) = \mathcal{O}\left(2^{i/5}k^{13/5}\right)$$

For convenient notation, set $\alpha = 2^{1/5}$ and $\beta = c_1 k^{13/5}$ for some constant c_1, so that the above expression becomes $\alpha^i\beta$. The total branching for picking planes at level i can thus be bounded by $\binom{\alpha^i\beta}{h_i}$. Taking the product of branching factors at each level gives the following (very similar to curve case):

$$\prod_{i=1}^{r} \binom{\alpha^i\beta}{h_i} \leq \prod_{i=1}^{r} \left(\frac{\alpha^i\beta}{\alpha^i h_0}\right)^{\alpha^i h_0} = \prod_{i=1}^{r} \left(\frac{\beta}{h_0}\right)^{\alpha^i h_0} = \left(\frac{\beta}{h_0}\right)^{\sum_{i=1}^{r} h_i} = \left(\frac{\beta}{h_0}\right)^{\varepsilon(k-K_r)}$$

The number of γ_{i+1}-rich lines at level i is $\frac{(k\gamma_{i-1})^2}{\gamma_{i+1}^3} = \frac{2k^2}{\gamma_i}$. From these we pick ℓ_i lines, giving a branching of $(\frac{k^2}{\gamma_i\ell_i})^{\ell_i}$ (up to constants in the base). The number of points at level j where $\gamma_j = \gamma_i^{4/5}$ is $k\gamma_i^{4/5}$, thus matching ℓ_i lines with this many points yields a branching of $(k\gamma_i^{4/5})^{\ell_i}$. Combining this with the branching factor above gives

$$(k\gamma_i^{4/5})^{\ell_i} \cdot \left(\frac{k^2}{\gamma_i\ell_i}\right)^{\ell_i} = \left(\frac{\alpha^i\beta}{\ell_i}\right)^{\ell_i}$$

By the same technique as the curves and planes, the total branching on lines can thus be bounded by

$$\left(\frac{\beta}{\ell_0}\right)^{\sum_{i=1}^{r} \ell_i} = \left(\frac{\beta}{\ell_0}\right)^{(1-\varepsilon)(k-K_r)}$$

Similar to the way the value h_0 is lower bounded in the proof of Lemma 3.22 in Section 3.6, the numbers h_0 and ℓ_0 can be lower bounded by $\Omega\left(\frac{\varepsilon(k-K_r)\log^{1/5} k}{k^{2/5}}\right)$ and $\Omega\left(\frac{(1-\varepsilon)(k-K_r)\log^{1/5} k}{k^{2/5}}\right)$. Let c_2 be the constant that collects the implicit constant in these lower bounds, the ignored constants in the base, and c_1. The total branching can now be bounded as follows:

$$\left(\frac{\beta}{h_0}\right)^{\varepsilon(k-K_r)} \cdot \left(\frac{\beta}{\ell_0}\right)^{(1-\varepsilon)(k-K_r)} =$$

$$\left(\frac{c_2k^3}{\varepsilon(k-K_r)\log^{1/5} k}\right)^{\varepsilon(k-K_r)} \cdot \left(\frac{c_2k^3}{(1-\varepsilon)(k-K_r)\log^{1/5} k}\right)^{(1-\varepsilon)(k-K_r)} =$$

$$\left(\frac{c_2k^3}{\varepsilon^\varepsilon(1-\varepsilon)^{1-\varepsilon}(k-K_r)\log^{1/5} k}\right)^{(k-K_r)} \leq \left(\frac{2c_2k^3}{(k-K_r)\log^{1/5} k}\right)^{(k-K_r)}$$

The Inclusion-Exclusion part runs in $2^{\frac{1}{2}K_r \log k} = \sqrt{k}^{K_r}$. By the same arguments as before, we can bound the total running time as desired. □

Lemma 3.38. *The polynomial time dependency of* PlaneCover *is* $\mathcal{O}\left(k^4 \log^4 k\right)$ *and its space complexity is* $\mathcal{O}\left(k^6 \log^2 k\right)$ *bits.*

Proof. Inclusion-Exclusion runs in $\mathcal{O}\left(n^{d+1}2^n\right)$ time when $n \leq k \log k$; the polynomial dependency is $\mathcal{O}((k \log k)^4)$. At any point there are at most $\binom{n}{d}$ candidates, so any internal node stores a set of at most $\mathcal{O}((k^2)^3)$ elements. There are at most $\mathcal{O}(\log k)$ such sets in memory at any time so $\mathcal{O}\left(k^6 \log^2 k\right)$ bits are enough to store them. □

3.6 Discussion

We have presented a general algorithm that improves upon previous best algorithms for all variations of *Curve Cover* as well as for the *Hyperplane Cover* problem in $\mathbb{R}^3$. Given good incidence bounds it should not be difficult to apply this algorithm to more geometric covering problems. However, such bounds are difficult to obtain in higher dimensions and for *Hyperplane Cover* the bound $\mathcal{O}\left(n^d/\gamma^3\right)$ is tight when no constraints are placed on the input, but it is too weak to be used even in $\mathbb{R}^3$. The bound by Elekes and Tóth works when the hyperplanes are well saturated, but the convenient relationship between saturation and degeneracy on hyperplanes does not extend past the $\mathbb{R}^3$ setting. Our hyperplane kernel guarantees a bound on the number of points on any j-flat. This overcomes the worst-case constructions for known incidence bounds, which involve placing very many points on the same line. An incidence bound for a kernelized point set might provide the needed foundation for similar *Hyperplane Cover* algorithms in higher dimensions.

A Algorithms

Algorithm 1 Recursive Curve Cover

procedure CC-RECURSIVE($P, \langle k_1, \ldots, k_r \rangle, i$)
  if $|P| > K_i s k / 2^{i-1}$ **then**
    return no
  if $|P| < K_i \log k$ **then**
    return INCLUSION-EXCLUSION(P, k')
  let S be the set of γ_i-rich-γ_{i-1}-poor candidates
  for all S' s.t. $S' \subseteq S, |S'| = k_i$ **do**
    if CC-RECURSIVE($P \setminus (\bigcup S'), \langle k_1, \ldots, k_r \rangle, i+1$) **then**
      return yes
  return no

Algorithm 2 Recursive Plane Cover

procedure PC-RECURSIVE($P, \mathcal{L}, \langle h_1, \ell_1, \ldots, h_r, \ell_r \rangle, i$)
  if $|P| > (K_i + |\mathcal{L}|)\gamma_{i-1}$ **then**
    return no
  if $|P| < K_i \log k$ **then**
    return INCLUSION-EXCLUSION($P \cup \mathcal{L}, K_i + |\mathcal{L}|$)
  let A be the set of lines in $\mathcal{L}$ added at depth $\left\lceil \frac{5(i-1)-2\log k}{4} \right\rceil$ or earlier
  if $|A| \geq 1$ **then**
    for all sets of $|A|$ planes $\mathcal{H}$ s.t. each $h \in \mathcal{H}$ covers a $\ell \in A$ and a $p \in P$ **do**
      if PC-RECURSIVE($P \setminus (\bigcup \mathcal{H}), \mathcal{L} \setminus A, \langle h_1, \ell_1, \ldots, h_r, \ell_r \rangle, i$) **then**
        return yes
    return no
  let H be the set of not-too-degenerate planes
  let L be the set of $\gamma_i\delta_i$-rich γ_{i-1}-poor lines
  for all $\langle H', L' \rangle$ s.t. $H' \subseteq H, |H'| = h_i$ and $L' \subseteq L, |L'| = \ell_i$ **do**
    let P' be the set of points that are in P but not on any $h \in H'$ or on any $\ell \in L'$
    if PC-RECURSIVE($P', \mathcal{L} \cup L', \langle h_1, \ell_1, \ldots, h_r, \ell_r \rangle, i+1$) **then**
      return yes
  return no

[2]

[2]Fixed typo in Algorithm 1 ($|S'|$ rather than $|S|$)

B Proof of Lemma 3.22

This entire section is used to prove Lemma 3.22. To do so, we must first give a number of auxiliary lemmas. The whole setup of the algorithm is to be able to use the incidence bound from Lemma 3.20 to bound the number of candidates at recursion level i. With a bound on the number of candidates we can bound the branching at level i as follows.

Lemma 3.39. *For some constant $c_1 = c_1(d, s)$ and $\alpha = 2^{d-1}$. The branching factor of an internal node of $\mathcal{T}$ at level i is bounded by $\left(\frac{\alpha^i c_1 k}{k_i}\right)^{k_i}$.*

Proof. Let the budget partition $\langle k_1, \ldots, k_r \rangle$ be fixed and consider recursion level i. At this point at most $K_i\gamma_{i-1} \leq sk^2/2^{i-1}$ points remain and all candidate curves in S are γ_i-rich. By Lemma 3.20, $|S|$ is bounded by one of the following:

$$\mathcal{O}\left(\frac{(K_i\gamma_i)^d}{\gamma_i^{2d-1}}\right) = \mathcal{O}\left(\left(\frac{sk^2}{2^{i-1}}\right)^d \left(\frac{sk}{2^i}\right)^{-(2d-1)}\right) \qquad = \mathcal{O}\left(\alpha^i s^{1-d} 2^d k\right)$$

$$\mathcal{O}\left(\frac{(K_i\gamma_i)}{\gamma_i^{2d-1}}\right) = \mathcal{O}\left(\left(\frac{sk^2}{2^{i-1}}\right) \left(\frac{sk}{2^i}\right)^{-1}\right) \qquad = \mathcal{O}(k)$$

Let c_1 be the smallest constant (dependent on the constants s and d) such that $\alpha^i \frac{c_1}{e} k$ is always greater than the implicit functions of both bounds. At level i, the algorithm will branch on all possible ways of picking k_i curves out of $|S| \leq \alpha^i \frac{c_1}{e} k$ candidates. We can bound this by

$$\binom{\alpha^i \frac{c_1}{e} k}{k_i} \leq \left(\frac{e\alpha^i \frac{c_1}{e} k}{k_i}\right)^{k_i} = \left(\frac{\alpha^i c_1 k}{k_i}\right)^{k_i}$$

□

For the worst case analysis, we need to know for which budget partition the product of branching factors is maximized. We therefore prove the following.

Lemma 3.40. *Let $k_1, \ldots, k_t$ be non-negative integers with a fixed sum, and $\alpha > 1$ and β constant real numbers. It holds that:*

$$\Pi = \prod_{i=1}^{t} \left(\frac{\alpha^i \beta}{k_i}\right)^{k_i} \leq \left(\frac{\beta}{k_0}\right)^{\sum_{i=1}^t k_i} \quad \text{where } k_0 = \frac{\alpha - 1}{\alpha^t - \alpha} \sum_{i=1}^{t} k_i$$

Proof. Let $\sum_{i=1}^t k_i = k$ and assume that $k_1, ..., k_t$ maximize Π. To prove the statement we explicitly compute the value of k_i as a function of α, i, r, and k. Let k_i and k_{i+1} sum to κ, and let c be any constant. Consider the function $f : [0, \kappa] \mapsto \mathbb{R}$, where $f(x) = \left(\frac{c}{x}\right)^x \left(\frac{\alpha c}{\kappa - x}\right)^{\kappa - x}$. Because k_i and k_{i+1} maximize Π, the function f is maximal at $f(k_i)$ and thus we derive the maximum of f by finding the maximum of its derivative.

$$\begin{aligned}
\log f(x) &= x(\log c - \log x) + (\kappa - x)(\log \alpha c - \log(\kappa - x)) \\
(\log f(a))' &= (\log c - \log x) - 1 - (\log \alpha c - \log(\kappa - x)) + 1 \\
&= \log \frac{\kappa - x}{x\alpha} = 0
\end{aligned}$$

From this we derive that $f(x)$ is maximal when $\kappa - x = x\alpha$, and thus that $\alpha k_i = k_{i+1}$. Since the above argument holds for all $i \geq 1$, only k_1 can be freely set and all other k_i are of the

form $\alpha^{i-1}k_1$. Define $k_0 = \alpha k_1$, so that for all $i \geq 1$ we have $k_i = \alpha^i k_0$. The expression for k_0 as it appears in the lemma can be derived from $\sum_{i=1}^{r} \alpha^i k_0 = k$ by finding the correct geometric series. Filling in the computed values for k_i in the definition of Π we get:

$$\Pi = \prod_{i=1}^{t} \left(\frac{\alpha^i \beta}{\alpha^i k_0} \right)^{\alpha^i k_0} = \prod_{i=1}^{t} \left(\frac{\beta}{k_0} \right)^{\alpha^i k_0} = \left(\frac{\beta}{k_0} \right)^{\sum_{i=1}^{t} k_i}.$$

□

The product Π here is essentially the bound on L from Lemma 3.22 that we are looking for. Before we can apply it however, we need to solve for k_0. Note that k_0 depends on r (the deepest recursion level of the algorithm) and the sum $\sum_{i=1}^{r} k_i$ (i.e. the budget used in the recursive part). To determine the deepest recursion level, recall that the algorithm keeps recursing until either too few or too many points remain. That means that we can derive the maximal recursion depth by solving what the recursion depth is where both those bounds are equal (i.e. no more branching can occur). The algorithm switches no later than depth r, where at most $\frac{(d-1)}{2} K_r \log k$ points remain. Conversely, if the instance was not immediately rejected, at most $K_t sk/2^{r-1}$ points remain. By solving for r and using the expression for k_0 from Lemma 3.40 we can prove the following.

Lemma 3.41. *Let r be the deepest level of recursion in* CC-Recursive*, k_0 is bounded by:*

$$k_0 = \mathcal{O}\left(\frac{(k - k_r)((d-1)\log k)^{d-1}}{(2sk)^{d-1}} \right).$$

Proof. The algorithm does not recurse if either the number of points left is less than $\frac{(d-1)}{2} k_r \log k$, or more than $k_r sk/2^{r-1}$. This means that it cannot recurse if

$$\frac{(d-1)}{2} k_r \log k > k_r sk/2^{r-1}$$

Setting these quantities equal and solving for r will thus give an upper bound on the recursion depth of any branch.

$$\begin{aligned} \frac{(d-1)}{2} k_r \log k &= k_r sk/2^{r-1} \\ 2^r &= \frac{4sk}{(d-1)\log k} \\ r &= \log \frac{k}{\log k} + \log \frac{4s}{d-1} \end{aligned}$$

Thus at most the budgets $k_1, ..., k_{r-1}$ summing to $k - k_r$ can be used by the recursive part of the algorithm. Plugging these values into Lemma 3.40 yields:

$$k_0 = \frac{k - k_r}{\sum_{i=1}^{r-1} \alpha^i} = \frac{(k - k_r)(\alpha - 1)}{\alpha^r - \alpha}.$$

We now expand $(2^{d-1})^r$

$$(2^{d-1})^r = (2^{d-1})^{\log \frac{k}{\log k} + \log \frac{4s}{d-1}} = \left(\frac{4sk}{(d-1)\log k} \right)^{d-1} = 2^{d-1} \left(\frac{2sk}{(d-1)\log k} \right)^{d-1}.$$

We substitute $\alpha^r = (2^{d-1})^r$ in the expression for k_0 to get:

$$k_0 = \frac{(k-k_r)(2^{d-1}-1)}{2^{d-1}\left(\frac{2sk}{(d-1)\log k}\right)^{d-1} - 2^{d-1}} = \Theta\left(\frac{(k-k_r)}{\left(\frac{2sk}{(d-1)\log k}\right)^{d-1} - 1}\right) = \Omega\left(\frac{(k-k_r)}{\left(\frac{2sk}{(d-1)\log k}\right)^{d-1}}\right).$$

By simplifying the last expression the proof is complete. □

We now have enough machinery to prove Lemma 3.22.

▶ **Lemma 3.22.** *Let L be the number of leaves in $\mathcal{T}_j$. Then for some constant $c_2 = c_2(d,s)$, L is bounded by*

$$L \leq \left(\frac{c_2 k^d}{(k-k_r)\log^{d-1} k}\right)^{K_j - k_r}$$

Proof of Lemma 3.22. Let $\alpha = 2^{d-1}$. Lemma 3.39 gives a bound of $\left(\frac{\alpha^i c_1 k}{k_i}\right)^{k_i}$ on the branching of an internal node at recursion level i. Taking the product of branching factors on the recursion levels j through r gives a bound on T_j. We can directly apply the bound from Lemma 3.40 to bound this product and therefore bound T_j.

$$T_j \leq \prod_{i=j}^{r-1}\left(\frac{\alpha^i c_1 k}{k_i}\right)^{k_i} \leq \left(\frac{c_1 k}{k_0}\right)^{\sum_{i=j}^{r-1} k_i} = \left(\frac{c_1 k}{k_0}\right)^{K_j - k_r}$$

Now substitute k_0 from Lemma 3.41 and collect any constants in $c_2 = c_2(d,s)$ to get

$$T \leq \left(\frac{c_1 k (2sk)^{d-1}}{(k-k_r)((d-1)\log k)^{d-1}}\right)^{K_j - k_r} = \left(\frac{c_2 k^d}{(k-k_r)\log^{d-1} k}\right)^{K_j - k_r}.$$

□

Chapter 4

A simple greedy algorithm for dynamic graph orientation

Edvin Berglin and Gerth Stølting Brodal

Graph orientations with low out-degree are one of several ways to efficiently store sparse graphs. If the graphs allow for insertion and deletion of edges, one may have to *flip* the orientation of some edges to prevent blowing up the maximum out-degree. We use arboricity as our sparsity measure. With an immensely simple greedy algorithm, we get parametrized trade-off bounds between out-degree and worst case number of flips, which previously only existed for amortized number of flips. We match the previous best worst-case algorithm (in $\mathcal{O}(\log n)$ flips) for general arboricity and beat it for either constant or super-logarithmic arboricity. We also match a previous best amortized result for at least logarithmic arboricity, and give the first results with worst-case $\mathcal{O}(1)$ and $\mathcal{O}(\sqrt{\log n})$ flips nearly matching degree bounds to their respective amortized solutions.

4.1 Introduction

An important building block in algorithmic theory and practice is the ability to store graphs with low memory usage and fast query times. Classical storage methods are edge lists and adjacency matrix, but both have pitfalls for *sparse* graphs: adjacency matrices use too much memory, while edge lists can have slow adjacency queries and/or updates on high-degree vertices. Much research has been devoted to improving these simple methods. The graph parameter *arboricity* α is a well-known measure of a graph's sparsity, which captures the minimum number of forests the edges of a graph can be partitioned into. Kannan et al. [22] showed how to efficiently store static graphs with low arboricity and supporting fast ($\mathcal{O}(\alpha)$ time) adjacency queries in the worst case.

Brodal and Fagerberg [8] extended this idea to consider *dynamic* graphs, where edges may be arbitrarily inserted or deleted. If the arboricity of the graphs remains bounded by a constant α, the forest partitions may be forced to change due to the updates. The authors deal with this by considering the problem of orienting the edges of the dynamic graph as in [22], but by re-orienting ("flipping") edges as needed to maintain low out-degree. They gave a simple greedy algorithm and proved that its amortized number of flips was $\mathcal{O}(1)$-competitive to the number of flips made by any other algorithm – even if that other algorithm is afforded unlimited computational resources and knowledge of the entire sequence of updates in advance. In this paper, we will use the term 'offline strategy' to describe such an algorithm. In particular, Brodal and Fagerberg showed how to maintain the out-degrees bounded by $\mathcal{O}(\alpha)$ with $\mathcal{O}(\log n)$ amortized flips, where n is the number of vertices in the graph. They also gave a lower bound of $\Omega(n)$ flips for maintaining the out-degrees bounded by α. It is not hard to see that this bound holds even for $\alpha = 1$.

Kowalik [24] gave another offline strategy and applied it to the algorithm by Brodal and Fagerberg, getting $\mathcal{O}(\alpha \log n)$ out-degree in constant amortized flips, demonstrating that a reasonable trade-off was possible. Both the algorithms of Brodal and Fagerberg [8] and Kowalik [24] need to know, and use as a parameter, a bound on the arboricity of the graph.

Kopelowitz et al. [23] later found a different algorithm, which came with slightly worse bounds but in the worst case rather than amortized. Their algorithm maintains $\mathcal{O}(\alpha + \log n)$ out-degree with $\mathcal{O}(\alpha + \log n)$ flips, without knowing α. However, if α is known, they give an alternate algorithm with somewhat faster running time but otherwise equal bounds. Also, if $\alpha = \mathcal{O}(\sqrt{\log n})$, both bounds can be improved slightly to $\mathcal{O}(\log n / \log\log n)$ due to some freedom in setting the base of the logarithmic terms.

He et al. [21] gave a new offline strategy with a parametrized trade-off between out-degree and flips, generalizing the two strategies in [8] and [24]. When applied to the algorithm by Brodal and Fagerberg it achieves $\mathcal{O}(\alpha\sqrt{\log n})$ out-degree with $\mathcal{O}(\sqrt{\log n})$ amortized flips. They also give another algorithm with worst-case bounds, nearly matching those in [23] but with somewhat simpler pseudocode.

Some other works, e.g. [14] and [29], solve the problem of maintaining dynamic graphs with a different approach, expressing their bounds in terms of the graph's h-*index*. The h-index of a graph is at least as large as the arboricity, but since the upper bound on this parameter is very crude ($\mathcal{O}(\sqrt{m})$ in a graph with m edges), it is difficult to compare these solutions.

The problem was originally motivated by quick adjacency queries [22]. But rather than making an explicit dictionary data structure, we focus on the problem of dynamically flipping edges to guarantee low maximum out-degree. This allows us to ignore lower bounds for dictionary operations, and we deliberately omit comparisons of update time complexity as they might be skewed unfairly in our favor. It is straightforward to create such a data structure on top of our machinery, should one so desire, by extending our solution to report which edges are

Table 4.1: Previous and new results for the dynamic edge orientation of dynamic graphs with bounded arboricity α. Flip bounds are either amortized (am.) or worst-case (w.c.) per update.

Reference	Out-degree	Flips	α known	Note
Brodal & Fagerberg [8]	$\mathcal{O}(\alpha)$	$\mathcal{O}(\log n)$ am.	yes	$\Omega(n)$ worst-case flips
Kowalik [24]	$\mathcal{O}(\alpha \log n)$	$\mathcal{O}(1)$ am.	yes	uses alg. from [8]
Kopelowitz et al. [23]	$\mathcal{O}(\alpha + \log n)$	$\mathcal{O}(\alpha + \log n)$ w.c.	no	
Kopelowitz et al. [23]	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ w.c.	no	if $\alpha = \mathcal{O}(\sqrt{\log n})$
He et al. [21]	$\mathcal{O}(\alpha\sqrt{\log n})$	$\mathcal{O}(\sqrt{\log n})$ am.	yes	uses alg. from [8]
He et al. [21]	$\mathcal{O}(\alpha \log n)$	$\mathcal{O}(\alpha \log n)$ w.c.	no	
New (Corollary 4.18)	$\mathcal{O}(\alpha + \log n)$	$\mathcal{O}(\log n)$ w.c.	no	
New (Corollary 4.19)	$\mathcal{O}(\alpha \log n)$	$\mathcal{O}(\sqrt{\log n})$ w.c.	no	
New (Corollary 4.20)	$\mathcal{O}(\log n)$	$\mathcal{O}(\alpha\sqrt{\log n})$ w.c.	yes	if $\alpha = \mathcal{O}(\sqrt{\log n})$
New (Corollary 4.17)	$\mathcal{O}\left(\alpha \log^2 n\right)$	$\mathcal{O}(1)$ w.c.	no	
New (Corollary 4.16)	$\mathcal{O}\left(\frac{\alpha \log^2 n}{f(n)}\right)$	$\mathcal{O}(f(n))$ w.c.	no	if $f(n) = \mathcal{O}(\log n)$

flipped. This allows programmers to tailor the balance between update and query time to suit their own needs.

Dynamic edge orientations have recently become a very popular building block in dynamic graph algorithms, especially for maintaining maximal matchings; see e.g. [4] [5] [31] [33] [34]. For an overview of other applications, we refer to the appendix in the full version of [23].

Our contribution

We present a new algorithm for maintaining an edge orientation of a dynamic graph, with a guarantee of low out-degree and worst-case number of flips. Like many previous solutions we relate the performance to the arboricity α of the dynamic graph, but unlike some previous works, ours does not require knowledge of the arboricity in the general case. Our algorithm is furthermore much simpler than previous ones, and uses queues as the only under-the-hood data structure. It owes its simplicity to the fact that it greedily chooses which edge to flip.

By controlling a run-time parameter, our algorithm allows a user-specified trade-off between the out-degree and the number of flips; this was previously only possible for algorithms with *amortized* number of flips. Depending on the choice of the parameter, the algorithm can maintain e.g. $\mathcal{O}(\alpha + \log n)$ out-degree with $\mathcal{O}(\log n)$ flips, or $\mathcal{O}\left(\alpha \log^2 n\right)$ out-degree with constant flips. Various other parameter settings are possible. We match or improve all known bounds with worst-case flips, except when the arboricity is within a specific, very narrow range.

4.2 Preliminaries

The *arboricity* of a graph G is the smallest number t such that the edges of G can be partitioned into t forests. Several equivalent definitions are used throughout the literature. We use arboricity(G) to denote the arboricity of G. A graph G with bounded arboricity arboricity$(G) \leq \alpha$ is *sparse*: any induced subgraph of G on $n' \leq n$ vertices contains at most $(n' - 1)\alpha$ edges. Note that while bounded arboricity graphs have no dense neighbourhood they can still have vertices of arbitrarily high degree, e.g. stars have arboricity 1 but maximum degree $n - 1$.

We say that $\mathcal{G} = G_0, G_1, G_2, \ldots, G_t$ is an *edit-sequence of graphs* if for each $i > 0$ there exists some edge (u, v) s.t. either $G_i = G_{i-1} \cup \{(u, v)\}$ (update i is an *insertion*) or $G_i = G_{i-1} \setminus \{(u, v)\}$

(update i is a *deletion*). We typically assume $G_0 = \emptyset$. We say that $\mathcal{G}$ has bounded arboricity (by a number α), or that arboricity$(\mathcal{G}) \leq \alpha$, if arboricity$(G_i) \leq \alpha$ for every i.

An *orientation* of a graph G is a directed graph $\overline{G}$ with the same vertex and edge sets as G, but where an undirected edge $(u, v) \in G$ exists as the directed edge (u, v) or (v, u) in $\overline{G}$. We use $\deg(\overline{G})$ to denote the maximum out-degree of $\overline{G}$; it is a c-orientation if $\deg(\overline{G}) \leq c$. Any graph G with arboricity$(G) \leq \alpha$ has an α-orientation; to see this, partition the edges into α forests, pick an arbitrary root in every tree, and direct every edge towards the root of its respective tree.

We say that $\overline{\mathcal{G}} = \overline{G}_0, \overline{G}_1, \ldots, \overline{G}_t$ is a *sequence of orientations* of $\mathcal{G}$ if every $\overline{G}_i$ is an orientation of G_i. Similarly, $\overline{\mathcal{G}}$ is a c-orientation if every $\overline{G}_i$ is a c-orientation. A *flip* is a triple (i, v, u) such that (v, u) is an edge in $\overline{G}_{i-1}$ and (u, v) is an edge in $\overline{G}_i$.

An *offline c-orientation strategy* κ is some method that takes $\mathcal{G}$ and produces a c-orientation $\overline{\mathcal{G}}$. By abusing notation we will also use κ to refer to the $\overline{G}$ produced by κ.

An *online c-orientation algorithm* $\mathcal{A}$ is analogous to the offline strategy, except that it receives $\mathcal{G}$ as a stream and has only a single $\overline{G}_i$ stored in memory at any time. Hence, upon receiving *update* i, it produces $\overline{G}_i$ as a function of G_i and $\overline{G}_{i-1}$ and then forgets $\overline{G}_{i-1}$. We also say that $\mathcal{A}$ *maintains* an online c-orientation of $\mathcal{G}$.

We say that κ or $\mathcal{A}$ makes σ flips (in the worst case) if the number of flips between any two updates $i, i+1$ is at most σ, and that it makes σ *amortized* flips if after any update i the total amount of flips is at most σi.

Note the difference in wording: a *strategy* has access to the whole sequence $\mathcal{G}$ and produces the entire c-orientation at once, possibly using brute force. The online *algorithm* instead sees $\mathcal{G}$ as a stream of unknown length and, after every update i, produces only a single "current" orientation.

4.3 The algorithm

The algorithm takes an edit-sequence of graphs $\mathcal{G}$ as an online stream, and a positive integer parameter k. Each vertex v maintains a standard FIFO queue Q_v which holds all of its out-edges. On an insertion (deletion) update, orient the new edge arbitrarily (delete the edge via object reference) and then k times pick a vertex v with maximum out-degree and flip the first edge in Q_v. The book-keeping of out-degrees is trivial by using e.g. a degree-indexed array and a pointer to the maximum degree. We do not explicitly support queries. See pseudocode below for ease of reading.

Algorithm 3 Greedy flipping algorithm

```
procedure INSERTION(v, u)
    push (v, u) to Q_v
    K-FLIPS
procedure DELETION(v, u)
    remove (v, u) from Q_v
    K-FLIPS
procedure K-FLIPS
    for i = 1 to k do
        let v be a max out-degree vertex
        pop an edge (v, u) from Q_v
        push (u, v) to Q_u
```

4.4 Analysis

To show the efficiency of Algorithm 3, we will prove that its out-degree is competitive to an unknown offline strategy. For given $\mathcal{G}$ and k, let δ, σ and ε be values satisfying the following

Table 4.2: Potential by type and placement in queue.

	Front	Back
Good	$1+2\varepsilon$	$1-\varepsilon$
Bad (first 3δ)	1	$1+\varepsilon$
Bad (rest)	1	1

conditions: (i) there exists an offline δ-orientation strategy κ of $\mathcal{G}$ making at most σ flips in the worst case, (ii) $0 < \varepsilon \leq 1$, and (iii) $k \geq 1 + 1/\varepsilon + 2\sigma$.

Theorem 4.1. *Algorithm 3 maintains an online $\mathcal{O}(\delta + (\delta\varepsilon + 1)\log_2 n)$-orientation of $\mathcal{G}$ with k flips and in $\mathcal{O}(k)$ time.*

Note that Algorithm 3 is completely oblivious to the values of δ, σ and ε, as well as any graph properties of $\mathcal{G}$ itself. The number of flips, and hence the running time, in Theorem 4.1 is trivial from the pseudocode. The rest of this section is dedicated to proving the bound on the out-degree. While the proof is quite non-trivial, the roadmap thereof is easy. We will associate potentials on all edges, such that the potential of an edge depends on where it is stored. Then we show that the total potential cannot increase, unless the maximum out-degree is $\mathcal{O}(\delta)$ in which case the potentials do not matter. Finally we re-interpret the moving of potentials as a game, where even an adversary cannot concentrate more than $\mathcal{O}((\delta\varepsilon + 1)\log n)$ extra potential in any single vertex – this also (roughly) bounds the maximum out-degree.

For purposes of analysis, we consider each queue Q_v to be two queues, the *Front* F_v and *Back* B_v. Edges are always inserted into B_v, and extracted from F_v. If F_v is empty when an edge should be extracted from Q_v, simply swap the two queues (by renaming) and then continue. It should be trivially clear that this is equivalent to using a single queue. We say an edge was flipped *from* v and *to* u if it was removed from Q_v/F_v and inserted into Q_u/B_u.

To bound the maximum out-degree, we introduce potentials on the edges. At update i, we say that an edge in $\overline{G}_i$ is *good* if it has the same orientation as in $\kappa(G_i)$ and *bad* otherwise. Good edges have $1 + 2\varepsilon$ potential if they are in a Front queue and $1 - \varepsilon$ in a Back queue. Bad edges have potential 1, except for the first 3δ bad edges in any Back queue which have potential $1 + \varepsilon$. Let $p(v)$ be the sum of potentials of all edges stored in Q_v, $\hat{p}(\overline{G}) = \max_v p(v)$ and $P(\overline{G}) = \sum_v p(v)$. When we need to differentiate the potential of a vertex in a specific orientation $\overline{G}_i$, we use $p_i(v)$ to denote $p(v)$ at the time that the algorithm was storing $\overline{G}_i$.

Since Algorithm 3 does not know the values of δ or ε, it cannot determine the exact potential of a vertex. But as the following lemma shows, the out-degree of a vertex is a close approximation of its potential. We will prove the theorem by bounding the maximum potential of any vertex, which then implies a bound on its degree.

Lemma 4.2. *For any vertex v, $\deg(v) + 5\delta\varepsilon \geq p(v) \geq \deg(v) - \delta\varepsilon$.*

Proof. For the upper bound, all edges contribute a base 1 potential, accounting for the $\deg(v)$ term. Note that at most δ out-edges of v are good. If they are all placed in F_v, they contribute an extra $2\delta\varepsilon$. At most 3δ bad edges in B_v contribute an extra ε each, giving at most $5\delta\varepsilon$ extra potential in total.

For the lower bound, only good edges in B_v can contribute less than 1 potential. Again there are at most δ of these and they contribute ε less, giving at least $\deg(v) - \delta\varepsilon$ potential in the vertex. □

Let $\beta = 6\delta\varepsilon$ be the *resolution* of the system. The following states that the potential of the highest-degree vertex is not too far from the maximum potential of any vertex.

Lemma 4.3. *Let u be some vertex with maximum potential, and let v be some maximum out-degree vertex. Then $p(u) - p(v) \leq \beta$.*

Proof. By Lemma 4.2, the potential of v is at least $p(v) \geq \deg(v) - \delta\varepsilon$ and the potential of u is at most $p(u) \leq \deg(u) + 5\delta\varepsilon \leq \deg(v) + 5\delta\varepsilon$. Rearranging we get $p(u) - p(v) \leq \deg(v) + 5\delta\varepsilon - (\deg(v) - \delta\varepsilon) = 6\delta\varepsilon = \beta$. □

Lemma 4.4. *Assume a vertex v has an empty F_v and at least 4δ edges in B_v. Then swapping F_v and B_v does not increase $p(v)$.*

Proof. The Back queue contains at most δ good edges and at least 3δ bad edges, hence exactly 3δ bad edges carry an extra ε potential which is released when moving from B_v to F_v. This $3\delta\varepsilon$ potential is enough to raise the potential of all δ good edges from $1 - \varepsilon$ to $1 + 2\varepsilon$. Any surplus potential is lost. □

Lemma 4.5. *Let v have out-degree at least 4δ. Then flipping an edge from v releases at least ε potential.*

Proof. By Lemma 4.4 we can assume F_v is non-empty. Let (u, v) be the edge moved from F_v to B_u. Note that if the edge was previously good it is now bad, and vice versa. Hence its potential decreases either from $1 + 2\varepsilon$ to at most $1 + \varepsilon$, or from 1 to $1 - \varepsilon$. □

Lemma 4.6. *Let S be any suffix of the sequence of flips performed by Algorithm 3 after some update. Let $d = \deg(\overline{G})$ at the start of S. Then $\deg(\overline{G}) \leq d + 1$ after S.*

Proof. Note that flips can increase the maximum degree only if there are at least two vertices u, v with maximum degree, and the algorithm flips an edge incident on both of them. As soon as some vertex reaches degree $d+1$, it will be the only vertex of maximum degree and immediately fall down to degree d in the following flip. Consequently no sequence of flips can raise a second vertex to degree $d+1$, which is a necessary condition for raising any vertex to degree $d+2$. □

Lemma 4.7. *Let v be a vertex that had an edge flipped from it on update i. Then $\deg_{\overline{G}_i}(v) \geq \deg(\overline{G}_i) - 2$.*

Proof. Take the suffix S of flips that begins with the last flip from v. Before S, v had maximum out-degree d. After S, $d - 1 \leq \deg(v)$ and $\deg(\overline{G}_i) \leq d + 1$ by Lemma 4.6. □

Consider the algorithm as it receives an update i. We say that the currently stored graph $\overline{G}_{i-1}$ has *sufficient degree* if each of the k flips associated with update i is from a vertex with out-degree at least 4δ. Conversely, we say the graph $\overline{G}_{i-1}$ has *insufficient degree* if at least one of the k flips is from a vertex with out-degree less than 4δ.

Lemma 4.8. *If $\overline{G}_{i-1}$ has insufficient degree, then $\deg(\overline{G}_i) = \mathcal{O}(\delta)$.*

Proof. Since some edge was flipped from a vertex with out-degree $d < 4\delta$, it follows from Lemma 4.6 that $\deg(\overline{G}_i) \leq d + 1 \leq 4\delta$. □

Lemma 4.9. *If $\overline{G}_{i-1}$ has sufficient degree, then $P(\overline{G}_i) \leq P(\overline{G}_{i-1})$.*

Proof. Assume update i is an insertion. The new edge is inserted into a Back queue, and adds at most $1 + \varepsilon$ potential. The offline strategy κ makes at most σ flips, which causes σ stored edges to swap their classification ("renaming") from good to bad or vice versa. A Front edge that was bad increases potential from 1 to $1 + 2\varepsilon$, and a Back edge that was good increases from $1 - \varepsilon$ to 1 or $1 + \varepsilon$. The renaming can therefore increase the total potential by at most

$2\sigma\varepsilon$. Each flip frees ε potential by Lemma 4.5 and the assumption of sufficient degree, so the total potential does not increase as long as $k\varepsilon \geq 1 + \varepsilon + 2\sigma\varepsilon$. This is guaranteed by the choice of parameters.

If the update was instead a deletion, the flips still release $k\varepsilon$ potential while even less potential is inserted. □

Note that the potential of the system can increase on both insertion and deletion updates if the graph has insufficient degree, since we cannot rely on Lemma 4.4 to ensure that the potential of a vertex is well-behaved when flipping edges from it. Also note that if κ is *known* not to perform any flips on deletion updates, *no* potential gets added to the system and so our algorithm can also forgo flipping on deletions.

So far we have shown that either the maximum out-degree is $\mathcal{O}(\delta)$, or we have a non-increasing quantity of potential and the degree of each vertex is closely approximated by its own potential. We next bound the maximum out-degree via a *counter game*, disassociated from the actual graph orientation, played by an adversary whose goal it is to concentrate as much potential as possible in a single counter. Counter games have been explored previously, under various names, in e.g. [10] and [28]: typically they may be thought of as two-player games where the second player is benevolent. Our game is different because the lone player is instead restricted by the concept of resolution β.

Formally, the game is played by a single player on n counters $x_1, \ldots, x_n$. Each counter x_i will hold a non-negative real-valued *weight* $|x_i|$, and the sum of weights is a constant $\sum_i |x_i| = X$. Any such distribution of X on the n counters is called a *game configuration* C. Let $\hat{x} = \max_i |x_i|$ be the maximum weight at any time. The player can perform arbitrarily many iterations of the following three-step operation: (i) pick a counter x_i and a $c > 0$ such that $|x_i| - c \geq \max(0, \hat{x} - \beta - 2)$, (ii) remove c weight from x_i and (iii) add positive weights whose sum is c to any set of counters.

The player is therefore allowed to redistribute weight *to* arbitrary counters, but must take it in not-too-large chunks *from* counters that are within the resolution (here $\beta+2$) of the maximum counter. Before upper-bounding $\hat{x}$, we show that the player is powerful enough to simulate the movement of potentials by Algorithm 3. We say a game configuration C *dominates* a graph orientation $\overline{G}$ if $|x_j| \geq p(v_j)$ for every j.

Lemma 4.10. *Let i be an update such that $\overline{G}_{i-1}$ has sufficient degree. Let C be a game configuration that dominates $\overline{G}_{i-1}$. Then the player can reach a game configuration C' that dominates $\overline{G}_i$.*

Proof. We need to show that if some vertex gains potential (so its corresponding counter no longer dominates it), then we can safely take enough weight from other counters to fill that 'gap'. Keep in mind that the total potential does not increase (Lemma 4.9). Since the player is allowed to redistribute weight *to* any counter, we let the gaps be filled in arbitrary order and only show that enough weight can be taken *from* other counters to make up the difference. If $\hat{x} > \hat{p}(\overline{G}_{i-1})$ then greedily take weight from all counters greater than $\hat{p}(\overline{G}_{i-1})$ to get $\hat{x} = \hat{p}(\overline{G}_{i-1})$.

Let v_j be a vertex that had an edge flipped from it. Then its resulting out-degree is $\deg_{\overline{G}_i}(v_j) \geq \deg(\overline{G}_i) - 2$ (Lemma 4.7) and its potential is $p_i(v_j) \geq \deg_{\overline{G}_i}(v_j) - \delta\varepsilon \geq \deg(\overline{G}_i) - 2 - \delta\varepsilon$ (Lemma 4.2). Also by Lemma 4.2 the maximum potential in the system is $\hat{p}(\overline{G}_i) \leq \deg(\overline{G}_i) + 5\delta\varepsilon$. Hence the final potential of v_j is within $6\delta\varepsilon + 2 = \beta + 2$ of the maximum potential. As the rules of the counter game allow us to take weight up to $\beta + 2$ from the maximum counter, then however much potential v_j lost we can take at least the same amount of weight from its corresponding counter x_j.

Conversely, if a vertex loses potential but its resulting potential is not at least $\hat{p}(\overline{G}_i) - \beta - 2$, it must have lost that potential due to deletion or renaming rather than flipping. Its counter can safely be left untouched and still dominate the potential of the vertex.

Since the sum of potential decreases (by flipping) is at least as large as the sum of increases (for any reason) (Lemma 4.9), and for any vertex that lost potential by flipping we can remove at least as much weight from its counter, then we can redistribute enough weight to raise the too-low counters to again dominate their respective vertex potentials. The updated counters form a game configuration that dominates $\overline{G}_i$. □

Lemma 4.11. *Let $\overline{G}_a, \ldots, \overline{G}_b$ be any sequence of orientations such that $\overline{G}_i$ has sufficient degree for every $a \leq i \leq b$. Consider a game with starting configuration C_a that dominates $\overline{G}_a$, with $\hat{x} = \hat{p}(\overline{G}_a)$. Then the player can reach game configurations $C_a, \ldots, C_b$ where C_i dominates $\overline{G}_i$ for every $a \leq i \leq b$.*

Proof. For every $a < i \leq b$ iterate Lemma 4.10 on C_{i-1} to create C_i. □

We now let an adversary play the game, with the goal to increase $\hat{x}$ as much as possible. For simplicity we assume that every counter is raised to $\hat{x}$ as the starting configuration. For $j = -1, 0, 1, 2, \ldots$ let $\ell_j = X/n + j(\beta + 2)$ be *weight level* j. A counter x_i is *above level* j, or *above* ℓ_j, if $|x_i| \geq \ell_j$. Let $X_j = \sum_{i=1}^{n} \max(0, |x_i| - \ell_j)$ be the *weight above* ℓ_j, and $\overline{X}_j = X - X_j$ the *weight below* ℓ_j. We say a counter x_i *contributes* $\max(0, |x_i| - \ell_j)$ to X_j and $\min(|x_i|, \ell_j)$ to $\overline{X}_j$.

Lemma 4.12. *Let j be a weight level such that $\ell_j \leq \hat{x}$. Let the player make any sequence of moves that maintain the condition $\ell_j \leq \hat{x}$. Then X_{j-1} does not increase.*

Proof. Note that any counter x_i contributes $\min(|x_i|, \ell_{j-1})$ to $\overline{X}_{j-1}$. By assumption there will always be a counter x_k with $\ell_j \leq |x_k|$. Hence the resolution rule prevents the player from making any counter contribute less to $\overline{X}_{j-1}$ than it already does. Since X is a constant and $\overline{X}_{j-1}$ is non-decreasing, $X_{j-1} = X - \overline{X}_{j-1}$ is non-increasing. □

Since $\ell_0 = X/n$ is the average weight of all counters, it must always be the case that $\hat{x} \geq \ell_0$ and $X_{-1} \leq n(\beta + 2)$.

Lemma 4.13. *Let j be a weight level such that $\ell_j \leq \hat{x} \leq \ell_{j+1}$. Let the player make any sequence of moves that maintain the condition $\ell_j \leq \hat{x} \leq \ell_{j+1}$. Then $2X_j \leq X_{j-1}$.*

Proof. By Lemma 4.12, X_{j-1} is a non-increasing amount. Let x_i be any counter that will contribute some positive weight w to X_j. Since the player maintains that $\hat{x} \leq \ell_{j+1}$, no counter will be able to contribute more than $\ell_{j+1} - \ell_j = \beta + 2$ to X_j, i.e. $0 < w \leq \beta + 2$. Then x_i must contribute $w + \beta + 2$ to X_{j-1}. Hence any counter that contributes to X_j contributes at least twice as much to X_{j-1}, and $2X_j \leq X_{j-1}$. □

The player is therefore stuck in the following dilemma: once $\hat{x}$ reaches some level ℓ_j, only a bounded amount X_{j-1} of weight remains available to redistribute. But once $\hat{x}$ reaches ℓ_{j+1}, only the weight above ℓ_j will be possible to redistribute. Therefore, in order to concentrate as much weight as possible above ℓ_{j+2}, the player must first maximize X_j without any counter actually reaching above ℓ_{j+1}.

Lemma 4.14. *The player cannot increase $\hat{x}$ to $\ell_{1+\log_2 n}$.*

Proof. Assume $\hat{x} \geq \ell_{\log_2 n}$. By alternatingly iterating Lemma 4.12 and Lemma 4.13, the weight above $\ell_{\log_2 n}$ is $X_{\log_2 n} \leq \left(\frac{1}{2}\right)^{1+\log_2 n} X_{-1} = \frac{1}{2n} X_{-1} \leq \frac{1}{2n} n(\beta+2) < \beta+2$. Since the weight is strictly less than $\beta+2$, even concentrating all of it in a single counter is not enough to make that counter reach $\ell_{1+\log_2 n}$. Hence $\hat{x} < \ell_{1+\log_2 n}$. □

We are now ready to prove the out-degree part of Theorem 4.1.

Proof of Theorem 4.1. Either the graph orientation has insufficient degree and maximum out-degree $\mathcal{O}(\delta)$ (Lemma 4.8) or it has non-increasing potential (Lemma 4.9) which is dominated by a counter game where the starting weight of any counter is $\mathcal{O}(\delta)$ (Lemma 4.11). By Lemma 4.14, the maximum counter is $\hat{x} < \ell_{1+\log_2 n} = \mathcal{O}(\delta) + (1+\log_2 n)(\beta+2)$. By Lemma 4.2, $\deg(v) \leq p(v) + \delta\varepsilon$, and therefore any vertex has out-degree bounded by $\mathcal{O}(\delta) + (1+\log_2 n)(\beta+2) + \delta\varepsilon = \mathcal{O}(\delta + (\delta\varepsilon+1)\log_2 n)$. □

4.5 De-amortizing offline strategies

In the previous work by Brodal and Fagerberg [8], their amortized algorithm is shown competitive with an offline strategy with bounded amortized number of flips, and hence subsequently published strategies have focused on achieving good amortized bounds. However, for our algorithm analysis, we require an offline strategy with *worst-case* flips per update. In this section we show one way to de-amortize offline strategies. Our technique does not generalize to every offline strategy, but relies on the special structure inherent to the strategies of both [23] and [21]. These strategies partition the edit-sequence into blocks of consecutive updates, with some length λ. No flips occur within a block, only in the seams between two blocks. The amortized flip complexity of these strategies is therefore simply the maximum number of flips between two blocks, divided by the length λ of the preceding block.

Since no flips are allowed within a block, the strategy is required to find an orientation of the union of all graphs $G_i, \ldots, G_{i+\lambda-1}$ within a block. The maximum out-degree of the entire strategy is therefore upper bounded by the maximum out-degree of any oriented union-graph. Higher λ gives a less sparse union-graph, necessitating higher out-degree, but also allows for a better amortized flip complexity. The following theorem shows a simple way of de-amortizing strategies with this structure, by taking all the flips between two blocks and spreading them evenly over the updates in the later block.

Theorem 4.15. *Let κ be a δ-orientation strategy of $\mathcal{G}$ where, for arbitrary λ, any update with $\sigma\lambda$ flips is followed by at least $\lambda-1$ updates with no flips. Then there exists a 2δ-orientation strategy of $\mathcal{G}$ making σ flips in the worst case.*

Note that if the last block of flips is not followed by $\lambda-1$ updates due to $\mathcal{G}$ ending, then one can pad $\mathcal{G}$ to appropriate length by repeatedly inserting and removing a dummy edge after the end of $\mathcal{G}$. Also note that λ can vary within the same sequence – blocks do not need to be of uniform length.

Proof. Let i be an update where κ performs a set of $\lambda\sigma$ flips. Let F be the set of flipped edges. Let κ' be an offline strategy with the same edge orientations as κ except on updates $i, \ldots, i+\lambda-1$. On any insertion update $i, \ldots, i+\lambda-1$, let κ' orient the new edge in the same direction as κ. Furthermore, on each update $i, \ldots, i+\lambda-1$, κ' takes σ arbitrary edges in F, removes them from F, and flips them.

Then F will be empty after update $i+\lambda-1$, so $\kappa(G_{i+\lambda-1}) = \kappa'(G_{i+\lambda-1})$. At all times F forms a δ-orientation, since F is a subset of $\kappa(G_{i-1})$. Similarly, $\kappa'(G_j) \setminus F$ is δ-orientation for

every $i \le j \le i + \lambda - 1$, since they are a subset of $\kappa(G_j)$. Hence κ' is a 2δ-orientation. Finally, κ' performs at most σ flips per update between updates i and $i + \lambda - 1$; exhaustively perform the same transformation on all of κ for σ flips on any update. □

4.6 Discussion

With our two theorems proven, we can relate the algorithm to known offline strategies and achieve the following corollaries. In all of the following, $\mathcal{G}$ is an arbitrary edit-sequence with arboricity$(\mathcal{G}) \le \alpha$.

Kowalik [24] presents an offline $\mathcal{O}(\alpha \log n)$-orientation strategy making 1 amortized flip. Using Theorem 4.15 we can de-amortize it to an offline $\mathcal{O}(\alpha \log n)$-orientation strategy making 1 flip in the worst case, giving the following two corollaries.

Corollary 4.16. *For a positive function $f(n) = \mathcal{O}(\log n)$, Algorithm 3 maintains an $\mathcal{O}\Big(\frac{\alpha \log^2 n}{f(n)}\Big)$-orientation with $k = 3 + \lceil f(n) \rceil$ flips.*

Proof. Let $\delta = \mathcal{O}(\alpha \log n)$, $\sigma = 1$ and $\varepsilon = 1/f(n)$. Then the algorithm maintains out-degree $\mathcal{O}\Big(\alpha \log n + \Big(\frac{\alpha \log n}{f(n)} + 1\Big) \log n\Big) = \mathcal{O}\Big(\frac{\alpha \log^2 n}{f(n)}\Big)$. □

Corollary 4.17. *Algorithm 3 maintains an $\mathcal{O}\Big(\alpha \log^2 n\Big)$-orientation of $\mathcal{G}$ with $k = 4$ flips.*

Proof. Let $f(n) \equiv 1$ in Corollary 4.16. □

Corollary 4.17 is the first result with $\mathcal{O}(1)$ worst-case flips. Compared to [24] (with $\mathcal{O}(1)$ *amortized* flips), it incurs an extra $\mathcal{O}(\log n)$ factor on the out-degree, but avoids the $\Omega(n)$ worst-case flips which that algorithm can experience.

Brodal and Fagerberg [8] give an offline $\mathcal{O}(\alpha)$-orientation strategy with $\mathcal{O}(\log n)$ flips in the worst case. It only makes flips on insertion updates.

Corollary 4.18. *Algorithm 3 maintains an $\mathcal{O}(\alpha + \log n)$-orientation of $\mathcal{G}$ with $k = \mathcal{O}(\log n)$ flips.*

Proof. Let $\delta = \mathcal{O}(\alpha)$, $\sigma = \mathcal{O}(\log n)$ and $\varepsilon = 1/\log n$. Then the algorithm maintains out-degree $\mathcal{O}\Big(\alpha + \Big(\frac{\alpha}{\log n} + 1\Big) \log n\Big) = \mathcal{O}(\alpha + \log n)$. □

He et al. [21] give an offline $\mathcal{O}(\alpha\sqrt{\log n})$-orientation strategy making $\mathcal{O}(\sqrt{\log n})$ amortized flips, which we de-amortize using Theorem 4.15.

Corollary 4.19. *Algorithm 3 maintains an $\mathcal{O}(\alpha \log n)$-orientation of $\mathcal{G}$ with $k = \Theta(\sqrt{\log n})$ flips.*

Proof. Let $\delta = \mathcal{O}(\alpha\sqrt{\log n})$, $\sigma = \mathcal{O}(\sqrt{\log n})$ and $\varepsilon = 1/\sqrt{\log n}$. Then the algorithm maintains out-degree $\mathcal{O}\Big(\alpha\sqrt{\log n} + \Big(\frac{\alpha\sqrt{\log n}}{\sqrt{\log n}} + 1\Big) \log n\Big) = \mathcal{O}(\alpha \log n)$. □

Corollary 4.20. *Algorithm 3 maintains an $\mathcal{O}(\log n)$-orientation of $\mathcal{G}$ with $k = \mathcal{O}(\alpha\sqrt{\log n})$ flips, if $\alpha = \mathcal{O}(\sqrt{\log n})$.*

Proof. Let $\delta = \mathcal{O}(\alpha\sqrt{\log n})$, $\sigma = \mathcal{O}(\sqrt{\log n})$ and $\varepsilon = 1/\alpha\sqrt{\log n}$. Then the algorithm maintains out-degree $\mathcal{O}\Big(\alpha\sqrt{\log n} + \Big(\frac{\alpha\sqrt{\log n}}{\alpha\sqrt{\log n}} + 1\Big) \log n\Big) = \mathcal{O}(\log n)$. □

Corollary 4.19 is an improvement over [23] in the flip complexity for edit-sequences with arboricity bounded by a constant. For $\alpha = \mathcal{O}(\sqrt{\log n}/\log\log n)$, Corollary 4.20 matches or improves the flip complexity from [23], albeit with a slightly worse degree bound, and only if α is known. If α is both $\mathcal{O}(\sqrt{\log n})$ and $\omega(\sqrt{\log(n)}/\log\log n)$ we are narrowly outperformed by [23], by no more than an $\mathcal{O}(\log\log n)$ factor.

Corollary 4.19 also nearly matches the degree bound in [21] but with worst-case flips instead of amortized. Corollary 4.18 matches the bounds in [23] for general arboricity and improves on their flip complexity if $\alpha = \omega(\log n)$. Furthermore, if $\alpha = \Omega(\log n)$, Corollary 4.18 matches the amortized bounds from [8].

Reverse trade-off

Compared to an offline strategy, our analysis lends itself to a trade-off in one direction, getting (at most) an $\mathcal{O}(\log n)$ factor on the out-degree for a constant factor on the number of flips. It allows us to perform much fewer flips than in [23] at the price of weaker degree bounds. A trade-off in the opposite direction would also be highly desirable, achieving out-degree (closer to) $\mathcal{O}(\delta)$ by making $\Omega(\sigma)$ flips. We have only found a very weak such trade-off:

Lemma 4.21. *Algorithm 3 can maintain an $\mathcal{O}(\alpha)$-orientation of $\mathcal{G}$ with $k = \mathcal{O}(\alpha n)$ flips.*

Proof. Let $\delta = \mathcal{O}(\alpha)$ and $\varepsilon = 1$ (the value of σ is inconsequential). Then each edge holds between 0 and 3 potential. And since any G_i has at most αn edges (by definition of arboricity), the total potential is between 0 and $3\alpha n$. Furthermore each flip releases 1 potential from the system, contingent on the graph having sufficient degree (Lemma 4.5). Hence after performing at most $3\alpha n$ flips on any starting orientation $\overline{G_i}$, we must reach a state where the next flip does not release potential, contradicting Lemma 4.5, and so by Lemma 4.8 the graph has out-degree at most $4\delta = \mathcal{O}(\alpha)$ after all flips. □

Lemma 4.21 only matches the worst-case bound of the algorithm in [8], which has drastically better amortized performance. Hence it should not be used in practice. Still, we believe a stronger reverse trade-off is possible and conjecture the following:

Conjecture 4.22. *For some function f, Algorithm 3 maintains an online $\mathcal{O}\Big(\delta + \frac{\sigma+1}{f(k)}\delta\log n\Big)$-orientation of $\mathcal{G}$ with k flips and in $\mathcal{O}(k)$ time.*

Dynamic arboricity

Throughout the paper we have done all our performance analysis against a static arboricity bound, i.e. a bound on the greatest arboricity seen anywhere in the edit-sequence. An interesting issue arises if the sequence contains contiguous sub-sequences, of non-trivial length, with higher or lower arboricity than elsewhere in the sequence. Some previous algorithms, e.g. one of the algorithms in [23] and the non-amortized algorithm in [21], adapt to increasing and decreasing arboricity automatically.

Our analysis immediately adapts to sequences with increasing arboricity, since the analysis can be performed on any prefix (or contiguous sub-sequence) of $\mathcal{G}$. In the case of periods with lower arboricity than earlier in the sequence, our algorithm obeys the new arboricity if the maximum out-degree is already within that new bound. In other words, if the maximum out-degree is already bounded relative to the new arboricity, then it will remain so. However, if the arboricity falls enough that the current maximum out-degree *breaks* the new bounds, our analysis does not require the maximum out-degree to decrease accordingly. Intuitively, using a k strictly larger than $1 + 1/\varepsilon + 2\sigma$ (thus experiencing a net loss of total potential with every

update) should force the maximum out-degree to tend towards the updated degree bounds, similar to the proof of Lemma 4.21. However, we do not have a formal argument for this.

Open problems

For all known strategies that maintain out-degree δ with σ (amortized) flips, it holds that $\delta\sigma = \Omega(\alpha \log n)$ and most achieve $\delta\sigma = \Theta(\alpha \log n)$. Can one design a strategy with $\delta\sigma = o(\alpha \log n)$?

Chapter 5

Further results

5.1 Offline edit sequences

This section makes a general statement about the structure of amortized offline edit sequences. Since the structure is very nicely behaved, it is our hope that this result can be used as a foundation for a general de-amortization scheme, as well as a starting point for other amortized offline sequences that sacrifice maximum out-degree for better flip complexity.

Let $\mathcal{G} = G_0, G_1, \ldots$ be an edit sequence, and let δ be some fixed integer such that there exists a δ-orientation strategy of $\mathcal{G}$. This section will use several different strategies for the same edit sequence, and we use $\kappa(G_i)$ to describe the oriented version of G_i produced by a strategy κ. When an orientation strategy orients a newly inserted edge as (v, u), we say that v is the *target* of that insertion.

For a given δ-orientation $\overline{G}$ of a graph G, let $\mathcal{R}[\overline{G}]$ be the directed sub-graph of $\overline{G}$ where a directed edge $(v, u) \in \overline{G}$ is in $\mathcal{R}[\overline{G}]$ if and only if v has out-degree δ in $\overline{G}$. Note that any vertex in $\mathcal{R}[\overline{G}]$ has out-degree either 0 or δ. A vertex is called a *leaf* if it as out-degree 0 in $\mathcal{R}[\overline{G}]$. A *leaf-path* (from a vertex v) is a vertex-disjoint directed path in $\mathcal{R}[\overline{G}]$ from v to any leaf. If v is itself a leaf, then the empty path is the only leaf-path from v. Conversely, if v is not a leaf, then the empty path is not a leaf-path from v.

For a δ-orientation strategy κ, we use $\mathcal{R}_i^\kappa$ as shorthand for $\mathcal{R}[\kappa(G_i)]$, or $\mathcal{R}_i$ when κ is clear from context. For an update i, we say κ flips a leaf-path (from a vertex v) if it flips a set $K \subseteq \kappa(G_{i-1})$ on the update and some subset of K is a leaf-path from v in $\mathcal{R}_{i-1}$.

The following two lemmas essentially state that flipping a leaf-path is a 'necessary and sufficient operation' for maintaining a δ-orientation on insertion updates. In other words, not only must any δ-orientation strategy flip a leaf-path, but flipping nothing but a leaf-path is enough to maintain a δ-orientation.

Lemma 5.1. *On an insertion update, flipping a leaf-path from the target is sufficient to maintain a δ-orientation.*

Proof. Let i be an insertion update and let $\overline{G}$ be any δ-orientation of G_{i-1}. Create $\overline{G}'$ from $\overline{G}$ by orienting the new edge as (v, u) and flipping an arbitrary leaf-path from v in $\mathcal{R}[\overline{G}]$.

If v was a leaf in $\mathcal{R}[\overline{G}]$ then the empty path was flipped, i.e. no edges were flipped, and $\overline{G}' = \overline{G} \cup \{(v, u)\}$ is a δ-orientation.

Otherwise, let P be the flipped leaf-path, starting in v and ending in the vertex w. Flipping all edges in P increments the out-degree of w, decrements the out-degree of v, and does not change the out-degree of any other vertex. Since w was a leaf it had out-degree at most $\delta - 1$ in $\overline{G}$, and now has out-degree at most δ. Since v had out-degree δ and lost one out-edge (flipping)

but gained one out-edge (insertion), it still has out-degree δ. No other out-degrees have been changed from $\overline{G}$, so $\overline{G}'$ a δ-orientation. □

Lemma 5.2. *Let κ be a δ-orientation strategy and let i be an insertion update. Then κ flips a leaf-path from the target on update i.*

Proof. Let K be the set of directed edges, oriented as in $\kappa(G_{i-1})$, that are flipped by κ for update i. Let v be the target of the insertion. If v has out-degree 0 in $\mathcal{R}_{i-1}$, then the empty subset of K is a leaf-path from v and we are done. Thus assume that v has out-degree δ in $\mathcal{R}_{i-1}$. Equivalently v has out-degree δ in $\kappa(G_{i-1})$, and since v gains one out-edge (via insertion), κ must also flip at least of its δ other out-edges or $\kappa(G_i)$ is not a δ-orientation. Note that it does not flip the new edge (v, u) – then we would simply have said that the edge was inserted with the opposite orientation (u, v) instead. Hence κ flips an edge that is an out-edge of v in $\mathcal{R}_{i-1}$.

Partition K arbitrarily into a set of cycles and maximal paths, i.e. so that the partition does not include any two paths P_1, P_2 such that P_2 begins where P_1 ends. Note that flipping all edges of a directed cycle does not change the out-degree of any constituent vertex. Therefore K must contain a path that contains v. Similarly, flipping the edges of a directed path does not alter the out-degrees of any of the internal nodes, but decrements the out-degree of the first vertex and increments the out-degree of the last one. So if any maximal path in K ends in a non-leaf, then $\kappa(G_i)$ is not a δ-orientation. Similarly, if there is no maximal path that starts in v, then the flips do not decrease the out-degree of v and v would get out-degree at least $\delta + 1$ in $\kappa(G_i)$. Hence K contains a maximal path P that starts in v and ends in a leaf. If P is not itself a leaf-path from v, then it must contain edges that are not in $\mathcal{R}_{i-1}$. Let P' be the longest prefix of P that only uses edges from $\mathcal{R}_{i-1}$. By the previous paragraph, P' contains at least one edge. Then P' is a leaf-path from v. □

It is important to note here that while any κ must flip some leaf-path, we cannot say *which* leaf-path is flipped. Furthermore, κ may choose to flip any other edges arbitrarily at both insertion and deletion updates in addition to 'the necessary leaf-path' from Lemma 5.2. Indeed, since it is an offline algorithm that can anticipate future updates, it might make all flips 'in advance' to ensure that the necessary leaf-path is always the empty path. We now show that without loss of generality, we can assume that any κ *only* flips the necessary leaf-path and no other edges. This is accomplished by transforming an arbitrary offline strategy without loss in either out-degree or amortized flip complexity. This technique will involve grouping flips together to the same update which breaks the worst-case number of flips of the starting strategy. Hence it is only applicable to strategies with bounded amortized number of flips.

We will characterize an orientation strategy κ as a pair of sets I and F, i.e. $\kappa = (I, F)$. A flip is a triple (i, v, u) where (v, u) is an edge in $\kappa(G_{i-1})$ and (u, v) is an edge in $\kappa(G_i)$. An insertion is a triple (i, v, u) such that (v, u) is an edge in $\kappa(G_i)$ and neither (v, u) nor (u, v) are edges in $\kappa(G_{i-1})$. Then I and F are simply the set of insertions and flips, respectively. We omit the deletions of edges from these characterizations, since clearly an edge is equally deleted from any orientation strategy regardless of its orientation. Note that we will only be concerned with pairs of sets that truly represent an orientation strategy: e.g. these sets cannot include any triples (i, v, u) with $i < 0$ or $i > |\mathcal{G}|$. It should then be clear that two strategies $\kappa = (I, F)$ and $\kappa' = (I', F')$ produce the same sequence of orientations if and only if $I = I'$ and $F = F'$. This gives a convenient notation for manipulating a given strategy by replacing insertions and flips in their respective sets.

For a strategy $\kappa(I, F)$ and an i, let $F_i = \{(j, v, u) \mid (j, v, u) \in F, j = i\}$ be the set of flips associated with update i, and $F_{\leq i} = \{(j, v, u) \mid (j, v, u) \in F, j \leq i\}$ be the set of flips associated with updates up to i. The sets I_i and $I_{\leq i}$ are defined analogously as subsets of I. Similarly, $\mathcal{G}_i$ is the prefix $G_0, \ldots, G_i$ of $\mathcal{G}$.

For a strategy $\kappa = (I, F)$, we say a set of flips $S \subseteq F$ is *impairing* κ (or simply *impairing*) if it there is a $S_d \subseteq S$ such that $\kappa' = (I, F \setminus S \cup \{(i+1, v, u) \mid (i, v, u) \in S_d\})$ is a δ-orientation. We say the flips in S_d are *early*, while the flips in $S \setminus S_d$ are *superfluous*, and we also say that an individual flip $f \in F$ is impairing if f is a member of any impairing set. Then κ' is the *improved strategy* and was created by *delaying* the early flips (from i to $i+1$) and *removing* the superfluous flips.

Lemma 5.3. *Suppose a strategy $\kappa = (I, F)$ flips a directed cycle C on some update i. Then the flips of C are impairing.*

Proof. The key observation is that flipping all edges in a directed cycle does not influence the out-degrees of any of the constituent vertices. Let $j > i$ be the earliest next update that any of the edges in C are either deleted or flipped again. If no such j exists then the flips of C are superfluous: the edges will not be deleted or flipped again after i, and they contribute equally to the out-degree of all constituent vertices regardless of which direction the cycle is oriented. Thus removing the flips of C at update i from F yields a δ-orientation.

If $j > i + 1$ then κ neither deletes nor flips these edges on update $i + 1$; all the flips of the edges in C can be delayed to $i + 1$ so they are early.

Otherwise $j = i+1$. Suppose j is a deletion update which deletes the edge $(v, u) \in C$. Then the flip (i, v, u) is superfluous and the flips of the edges in $C \setminus \{(v, u)\}$ are early.

Otherwise, $j = i + 1$ is an update where all edges in C remain in the graph, and where κ re-flips some $C' \subseteq C$ of the edges. Then the flips of $C \setminus C'$ at update i are early, while the flips of C' at updates i and j are superfluous. □

Lemma 5.4. *For a δ-strategy $\kappa = (I, F)$, let i be an update where the edges flipped in F_i induce a connected component $C \in G_{i-1}$ such that update i did not insert a new edge with target in C. Then some flip in F_i is impairing.*

It is important here to note that this lemma applies to both insertion and deletion updates: for insertion updates it applies to every connected component except the component which includes the target, and for deletion updates it applies to every single connected component.

Proof. Assume for contradiction that F_i includes no impairing flip. Let $F_C \subseteq F_i$ be the flips which induce the connected component C, and let F_t be the (possibly empty) subset of F_i which contains the target of the insertion. If i was a deletion, then $F_t = \emptyset$. Then clearly $(I_{\leq i}, F_{\leq i} \setminus F_C)$ is a δ-strategy of $\mathcal{G}_i$ – so F_C is impairing for this prefix $\mathcal{G}_i$, but not for the entire $\mathcal{G}$. Let $j > i$ be the smallest number such that $F_i \setminus F_v$ contains an impairing flip for $(I_{\leq j-1}, F_{\leq j-1})$ but not for $(I_{\leq j}, F_{\leq j})$. By assumption this j exists, but note that the impairing set to which f belongs is not necessarily a subset of F_i.

Suppose $j > i+1$. Then the flips in $F_i \setminus F_t$ are early for $\mathcal{G}$. To see this, we know that the entire set $F_i \setminus F_t$ is superfluous for $\mathcal{G}_i$, and that it contains some impairing flip $f = (i, v, u) \in F_i \setminus F_t$ for $\mathcal{G}_{i+1}$. Being superfluous and early is not mutually exclusive, so it is sufficient to show that if f is superfluous it is also early. But this is obvious: if $f = (i, v, u)$ is superfluous for G_{i+1}, then the flips F_{i+1} do not require a specific orientation of the edge (v, u) to maintain a δ-orientation, and so we could instead delay f rather than remove it. This ensures that all subsequent flips for $i+2, \ldots$ also maintain δ-orientations. Hence $(I, F \setminus \{(i, v, u)\} \cup \{(i+1, v, u)\})$ is a δ-orientation strategy of $\mathcal{G}$, and F_i contained an impairing flip.

Otherwise $j = i + 1$. We get three sub-cases. Let $f = (i, v, u) \in F_i \setminus F_t$ such that j is a deletion update which deletes the edge (u, v). Then $\{f\}$ is superfluous in some impairing subset of F_i. If no such flip exists, let $f = (i, v, u) \in F_i \setminus F_t$ be a flip such that there exists a $f_2 = (i + 1, u, v) \in F_{i+1}$. Then $\{f, f_2\}$ are superfluous in some impairing subset of $F_i \cup F_{i+1}$:

the edge has orientation (v, u) in orientation of G_{i-1}, should also have orientation (v, u) in the orientation of G_{i+1}, and is allowed to have either orientation for G_i. If neither of the previous two sub-cases apply, then $F_i \setminus F_t$ is an early set: none of these edges are removed or flipped again on update $i+1$ and are allowed to have either orientation in the orientation of G_i. Hence we can delay all of them to $i+1$.

All four cases imply contradiction, so F_i includes an impairing flip. □

Let $\kappa = (I, F)$ be an arbitrary δ-orientation strategy of $\mathcal{G}$ that performs at most σ amortized flips. Let $\kappa^* = (I, F^*)$ be a δ-orientation strategy that is the result of exhaustively choosing an arbitrary impairing set in κ, and replacing κ with its improved strategy κ', until a strategy with no impairing sets is reached. Note that κ^* is not necessarily unique – the order of choosing impairing sets can lead to different fix-points.

Lemma 5.5. *κ^* is a δ-orientation strategy making at most σ amortized flips.*

Proof. Since κ is a δ-orientation and κ^* is formed from κ by operations that maintain this property, κ^* is a δ-orientation.

Assume for contradiction that κ^* makes more than σ amortized flips. Then there is prefix $\mathcal{G}_i$ of $\mathcal{G}$ such that κ^* makes $|F^*_{\leq i}| > i\sigma$ total flips on $\mathcal{G}_i$, while κ makes $|F_{\leq i}| \leq i\sigma$ total flips on $\mathcal{G}_i$ since it makes at most σ amortized flips.

But $F^*_{\leq i}$ was constructed by removing and delaying flips in $F_{\leq i}$: hence every flip in $F^*_{\leq i}$ can be uniquely associated with a flip in $F_{\leq i}$. Therefore $|F^*_{\leq i}| \leq |F_{\leq i}|$, contradiction. □

Lemma 5.6. *On any deletion update, κ^* flips no edges.*

Proof. Follows from Lemma 5.4, since there is no target on deletion updates. □

Lemma 5.7. *On any insertion update i, the edges induced by the flips F^*_i form a directed acyclic graph in $\kappa^*(G_{i-1})$, and a connected component in G_{i-1}.*

Proof. By Lemma 5.3, if F^*_i contains any cycle, all these flips are impairing. Similarly by Lemma 5.4, if F^*_i induced two or more components in G_{i-1}, some flip is impairing. But κ^* has no impairing flips by construction. □

Lemma 5.8. *On any insertion update i, κ^* flips one leaf-path from the target and no other edges.*

Proof. Consider an insertion $(i, v, u) \in I^*$. By Lemma 5.7, κ^* flips a connected DAG $D \subseteq \kappa^*(G_{i-1})$ on update i.

For a directed edge (w, x) in $\mathcal{D}$, we say that the edge is an *in-going flip* on x and an *out-going flip* from w. Every in-going flip increases the degree of the vertex by 1, while every out-going flip decreases it by 1.

Consider a vertex $w \neq v$ that has strictly more out-going flips than in-going flips. This decreases the out-degree of w from $\overline{G}_{i-1}$ to $\overline{G}_i$. Hence at least one of the out-going flips of w is early, contradicting the construction of κ^*. Conversely, if v had out-degree δ in $\kappa^*(G_{i-1})$, its *must* have strictly more out-going flips than in-going flips, to incur a net loss of out-degree which accommodates the newly inserted out-edge (v, u). However, if it has at least two more out-going flips than in-going flips, then some out-going flip from v is early. Consequently, if v had out-degree δ in $\kappa^*(G_{i-1})$, then v has exactly one more out-going than in-going flip in D.

Perform a topological sort on the vertex set of D. Since D is connected and v is the only vertex with more out-going than in-going edges, no other vertex can be sorted before v. Since v has is sorted first, it has no in-going flips, and therefore one out-going flip. By temporarily

hiding v and its out-going flip (v, u), the same argument holds for the neighbour u in $D \setminus \{u\}$ and so u must have exactly one out-going flip. So by exhaustively iterating this argument, we get that D is a path starting in v. The path is clearly vertex-disjoint (or it would contain a cycle), and from lemma Lemma 5.2 we know that it ends in a leaf (which may be v itself). Hence D is a leaf-path from v. □

An orientation strategy is said to be *nice* if it flips no edges on deletion updates, and flips only a leaf-path from the target on insertion updates.

Lemma 5.9. *κ^* is nice.*

Proof. Follows from Lemma 5.6 and Lemma 5.8. □

Theorem 5.10. *Suppose there exists δ-orientation of $\mathcal{G}$ making σ amortized flips. Then there exists a nice δ-orientation of $\mathcal{G}$ making σ amortized flips.*

Proof. κ^* is a δ-orientation of $\mathcal{G}$ making σ amortized flips, by Lemma 5.5, and it is also nice, by Lemma 5.9. □

5.2 Lossy counter games

One of the main difficulties in proving an improved bound for Algorithm 3 in Chapter 4, is that we have no statements of how the potential of the system behaves when k is strictly more than $1 + 1/\varepsilon + 2\sigma$, not even if the difference is quite pronounced. This section is one attempt to bridge that gap by generalizing the counter game in Chapter 4 to have a loss factor. That is to say, any time the player makes a move, he incurs some multiplicative loss on the weight he 'touches'. Unfortunately this idea seems hard to combine with the analysis of Algorithm 3 as-is, since the expected potential *lost* can be much less than the potential *moved*, even with $\varepsilon \approx 1$: most moves have a loss factor of $\frac{1-\varepsilon}{1+\varepsilon} \to 0$, but some moves have a loss factor of $\frac{1+\varepsilon}{1+2\varepsilon} \to 2/3$ and we cannot push this fraction any lower.

Furthermore, with the deterministic approach of Algorithm 3, it seems very hard to say for sure when moves will a high factor will occur. Even if we set k to be very much larger than $1 + 1/\varepsilon + 2\sigma$, it is unclear how to push the loss factor down to a guarenteed low value with the deterministic algorithm. However, it seems likely that with a slight modification to the greedy algorithm that adds *random choice* to which edge is popped, one would with high probability get a loss factor that is consistently low over a sub-sequence of updates. To this end, we design the *lossy* counter game. We expect the reader to be familiar with the notation from the lossless counter game, of which this game inherits a significant portion.

Let $x_0, x_1, \ldots, x_{n-1}$ be a set of n counters, with each x_i holding a non-negative real value (or *weight*) $|x_i|$. Let $\beta > 0$ be the *resolution* of the system and let ϕ be the *loss constant*, with $0 < \phi \leq 1$. It may be convenient to think of $1 - \phi$ as the *loss factor*. For the case that $\phi = 1$, the game does not suffer any loss and the results herein will be no different than the counter game in Chapter 4. [1]. For this reason we will assume that $\phi < 1$. Let $\hat{x}$ refer to $\arg\max_{x_i} |x_i|$, the current maximum weight in any counter.

A player may perform an arbitrary sequence of the following three-step operation: (i) let i and $c > 0$ be values such that $|x_i| - c \geq \max(\hat{x} - \beta, 0)$, (ii) decrease $|x_i|$ by c, and (iii) add positive weights, with sum at most ϕc, to any set of counters.

The player is therefore allowed to redistribute weight *to* any counter, but must take it in not-too-large chunks *from* counters that are within the resolution β of the maximum counter value. Furthermore, each time the player makes a move, he incurs a multiplicative loss of $1 - \phi$ on the weight that is relocated. The goal of the player is to maximize $\hat{x}$.

Given a game configuration C, we refer to any sequence S of moves starting from C as a *play*. An *optimal* play achieves the maximum increase of $\hat{x}$ among all possible plays, and an *optimal player* is one who performs any optimal play.

Conjecture 5.11. *The optimal play increases $\hat{x}$ by $\mathcal{O}(\beta\phi \log n + \beta)$.*

This proposed bound is not taken out of thin air, but the combination of the individual bounds by two simple different plays: $\mathcal{O}(\beta\phi \log n)$ when ϕ is decently large, and $\mathcal{O}(\beta)$ when ϕ is close to 0. The former is intuitively obvious as we already have this bound for $\phi = 1$, so $\phi \approx 1$ 'should' behave similarly. Any other play *seems* to be a combination of these, although we have been unable to prove this. However, the proofs for the bounds of the two types of play are found below.

We say that any counter x_i has $c_i = \max(0, \hat{x} - |x_i|)$ *available weight*, and that any counter with positive available weight is *alive* while those with 0 available weight are *dead*.

Although perhaps not immediately obvious, the player may choose to make moves that *lower* $\hat{x}$ depending on the starting condition, e.g. if the maximum counter is just a little more than β

[1] The referenced counter game uses the resolution $\beta + 2$ rather than β in order to comply with the notation of the rest of the paper. However, the game itself can easily adapt to any resolution, even in the lossless version.

above all other counters: then it is beneficial to remove weight from this maximum counter, in order to make a greater sum of weights available to redistribute.

Lemma 5.12. *Let S be an optimal play. After the first move in S which raises $\hat{x}$, S does not lower $\hat{x}$.*

Proof. Let S be any play, and let S' be the longest suffix of S such that the first move of S' increases $\hat{x}$. Then our task is to show that the optimal sub-play S' does not lower $\hat{x}$. But this is trivial: if $\hat{x}$ is lowered in S', the weight spent to raise $\hat{x}$ in the first move of S could instead have been spent on raising some dead counter, and have more total available weight at the same $\hat{x}$ level. □

For this reason we will only consider plays that have already performed their 'catching up' to make as much starting weight available as possible. As the lemma states, from that point the optimal player never lowers $\hat{x}$ again, and so for our purposes the play is said to start at this point. A *full play* is a play that ends with a single counter alive, i.e. exhausts all its available moves.

We keep all alive counters sorted, such that $|x_0| \geq |x_1| \geq \ldots |x_i|$ for $i+1$ alive counters. This is easy to ensure: if any move changes the order, simply re-label the alive counters. This implies that $\hat{x} = x_0$ at all times.

Lemma 5.13. *Without loss of generality, an optimal play does not add weight to a dead counter.*

Proof. By Lemma 5.12 the optimal play S will not $\hat{x}$. Let x_i be the counter from which weight is removed, and let x_k be a dead counter that receives weight. Let $c \leq c_i$ be the portion of c_i such that ϕc is added to x_k: the updated available weight on x_k is then then $c'_k \leq \phi c < c$. Let $d = c - c'_k$. Replace the move in question with the move that takes $c_i - c + d$ weight from x_i: distribute $\phi(c_i - c)$ as originally, and use the weight $\phi d \geq 0$ on a trivial move that adds to all alive counters including x_i. Then the available weight is exactly the same for every alive counter as in the original move (after re-labeling), furthermore $\hat{x}$ has been raised by an additional non-negative amount. □

A *trivial* move takes some weight c from an alive counter and distributes ϕc evenly across all other alive counters. We also say that 'part of a move' can be a trivial move, i.e. take weight c from an alive counter, for a $c' \leq c$ distribute $\phi c'$ evenly across all other alive counters, and distribute $\phi(c - c')$ in some other unspecified way. Note that the available weight on any other alive counter remains unchanged after a trivial move.

An *ordered move* takes all available weight c_i from the smallest alive counter x_i, and distributes ϕc_i among the remaining alive counters in some fashion. An *unordered* move is any move that is not ordered. An *ordered play* is a play in which every move is ordered.

Lemma 5.14. *Without loss of generality, an optimal play is an ordered play.*

Proof. Let S be an optimal play which is not ordered. Let S' be the smallest suffix of S such that the first move in S' is unordered. Let x_i be smallest alive counter in the starting configuration of S'. There are three cases.

Case 1. The player makes a move that raises $\hat{x}$ enough that x_i is no longer alive. Then S is not optimal: it is outperformed by the strategy that makes a trivial move from x_i before performing the moves S'.

Case 2. The player takes weight from a counter x_j, $j < i$, and places it in a way that does not raise $\hat{x}$. Then without loss of generality, there is an optimal play that makes an ordered move from x_i. In particular, since S' is the shortest unordered suffix of S, its next move is

ordered and takes all weight from x_i. Let $\phi c'$ be the weight placed on x_i in the move from x_j, which is possibly 0. If $\phi c' = 0$, the two counters do not interact, i.e. taking from them in the opposite order but redistributing their combined weight in the original order yields a play S'' which starts with an ordered move and is at least as good as S'. The second move in S'' is not necessarily ordered; recurse the proof on the strictly shorter suffix of S.

If $\phi c' > 0$, this quantity gets moved twice in the first two moves of S'. Then S' is not optimal: it distributes at most $\phi(c_i + \phi c')\phi(c_j - c') < \phi(c_j + c_i)$ among the counters $x_0, \ldots, x_{i-2}$. Perform the same transformation of S' as above and make a trivial move with the additional weight on the move from x_j.

Case 3. The player makes an unordered move from x_j that raises $\hat{x}$, but not so much that x_i is no longer alive. Let $c_i' < c_i$ be the updated available weight of x_i after the move, such that $\hat{x}$ was raised by $c_i - c_i'$. Then replace the first two moves in S' thusly: perform an ordered move on x_i that greedily fills the target counters of the original move from x_j. Since $c_j \geq c_i$, this move raises $\hat{x}$ by no more than the original first move in S'. Hence $c_j - c_j' \leq c_i - c_i'$, and the move from x_j can distribute at least as much weight as the original second move in S'. In the case of surplus weight ($c_j - c_j' \leq c_i - c_i'$), then S was not optimal since this extra weight can be used for a trivial move that increases $\hat{x}$ further. □

We define a *weak* move as the ordered trivial move, i.e. takes all available weight c_i from the smallest alive counter x_i, and distributes ϕc_i evenly among the remaining i alive counters.

Lemma 5.15. *Consider a full play of weak moves. Then the increase of $\hat{x}$ is $\mathcal{O}(\beta\phi \log n)$.*

Proof. Let S be a length i play of weak moves. Since weak moves are ordered trivial moves, the increase is $\sum_{j=i}^{1} \frac{1}{j}\phi c_j$. Since $i \leq n$ and $c_j \leq \beta$ for every j, the expression simplifies to $H_i \phi\beta = \mathcal{O}(\beta\phi \log n)$. □

A *strong* move takes all available weight c_i from the smallest alive counter i and puts ϕc_i on x_0. Since the move increases x_0 and no other counters, every other alive counter x_j sees an increased distance to $\hat{x}$ and their corresponding available weight c_j will decrease. For some i, j with $0 < j < i$, we use $c_j^{(i)}$ to denote the available weight in counter c_j immediately after the player has performed a strong move on counter x_i. For ease of notation, for a play of k strong moves we use $c_i^{(k+1)} = c_i$ is used to refer to the value of c_i before the play.

Lemma 5.16. *Consider a strong move from x_i, and let x_j be an alive counter with $0 < j < i$. Then $c_j^{(i)} = c_j^{(i+1)} - \phi c_i^{(i+1)} > 0$.*

Proof. The strong move increases x_0 by $c_i^{(i+1)}$, and increases no other counter. Hence the distance from x_j to $\hat{x}$ has increased by $c_i^{(i+1)}$, i.e. the weight available on c_j after the ith move has decreased by exactly $\phi c_i^{(i+1)}$.

The updated value $c_j^{(i)} > 0$ because $c_i^{(i+1)} \leq c_j^{(i+1)}$ and $\phi < 1$, hence $\phi c_i^{(i+1)} < c_j^{(i+1)}$. □

Lemma 5.17. *For a full play of i strong moves, $\hat{x}$ increases by $\phi\left(\sum_{r=1}^{i}(1-\phi)^{r-1} c_r^{(i+1)}\right)$.*

Proof. Since there are i strong moves and a move from x_r increases $\hat{x}$ by $\phi x_j^{(j+1)}$ for every $0 < r \leq i$. Hence the total increase is $\phi \sum_{r=1}^{i} c_r^{(r+1)}$, and we need to find the updated values $c_r^{(r+1)}$. We do this via induction on $j = 1, \ldots, i$. The case for $j = 1$ is obvious since $\phi c_1^{(2)} = \phi \sum_{r=1}^{1}(1-\phi)^{r-1} c_r^{(j+1)}$.

Now for the induction step, suppose the statement is true for a full play of $j-1$ moves, and show it also holds for j strong moves. Then the increase is

$$\phi c_j^{(j+1)} + \phi \left(\sum_{r=1}^{j-1} (1-\phi)^{r-1} c_r^{(j)} \right) =$$
$$\phi c_j^{(j+1)} + \phi \left(\sum_{r=1}^{j-1} (1-\phi)^{r-1} \left(c_r^{(j+1)} - \phi c_j^{(j+1)} \right) \right) =$$
$$\phi \left(\sum_{r=1}^{j-1} (1-\phi)^{r-1} c_r^{(j+1)} + c_j^{(j+1)} \left(1 - \phi \left(\sum_{r=1}^{j-1} (1-\phi)^{r-1} \right) \right) \right) =$$
$$\phi \left(\sum_{r=1}^{j-1} (1-\phi)^{r-1} c_r^{(j+1)} + c_j^{(j+1)} \left(1 - \phi \left(\frac{1-(1-\phi)^{j-1}}{1-(1-\phi)} \right) \right) \right) =$$
$$\phi \left(\sum_{r=1}^{j-1} (1-\phi)^{r-1} c_r^{(j+1)} + c_j^{(j+1)} (1-\phi)^{j-1} \right) =$$
$$\phi \left(\sum_{r=1}^{j} (1-\phi)^{r-1} c_r^{(j+1)} \right).$$

□

Lemma 5.18. *For a full play of strong moves, the increase of $\hat{x}$ is at most β.*

Proof. Let S be a length i full play of strong moves. From Lemma 5.17, the increase of $\hat{x}$ by S is $\phi \left(\sum_{r=1}^{i} (1-\phi)^{r-1} c_r^{(i+1)} \right) \leq \phi \left(\sum_{r=1}^{\infty} (1-\phi)^{r-1} \beta \right) = \phi \beta \frac{1}{\phi} = \beta$. □

Lemma 5.15 and Lemma 5.18 combine into the bound $\mathcal{O}(\beta\phi \log n + \beta)$, as stated in Conjecture 5.11.

Postface

This version of the thesis has been edited slightly from the original version. The changes are:

- Added bibliography item for the publication *A simple greedy algorithm for dynamic graph orientation* [3], and
- edited one sentence in the Preface to reflect that this paper has now been published.
- Fixed a reference to Lemma 5.2 (used to say "Lemma XXX") in the proof for Lemma 5.8.
- Fixed an occurrence of $\mathcal{O}(\log n)$ (used to say "$\mathcal{O}(\log)$"), in the abstract for Chapter 4.
- Renamed Chapter 5 to *Further results*, from *Unpublished work*.
- Added the word 'can' to the final sentence of Section 2.2, to reflect that this statement is unproven speculation.
- Added this Postface.

Bibliography

[1] Peyman Afshani, Edvin Berglin, Ingo Van Duijn, and Jesper Sindahl Nielsen. Applications of incidence bounds in point covering problems. *arXiv preprint arXiv:1603.07282*, 2016. v

[2] Pankaj K Agarwal and Boris Aronov. Counting facets and incidences. *Discrete & Computational Geometry*, 7(1):359–369, 1992. 31, 32

[3] Edvin Berglin and Gerth Stølting Brodal. A Simple Greedy Algorithm for Dynamic Graph Orientation. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th International Symposium on Algorithms and Computation (ISAAC 2017)*, volume 92 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-054-5. doi: 10.4230/LIPIcs.ISAAC.2017.12. URL `http://drops.dagstuhl.de/opus/volltexte/2017/8263`. v, 65

[4] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. *arXiv preprint arXiv:1506.07076*, 2015. 45

[5] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 692–711. Society for Industrial and Applied Mathematics, 2016. 45

[6] Smitri Bhagat, Moira Burke, Carlos Diuk, Ismail O Filiz, and Sergey Edunov. Three and a half degrees of separation, 2016. URL `http://bit.ly/2hapT0N`. Accessed September 10, 2017. 13

[7] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set partitioning via inclusion-exclusion. *SIAM Journal on Computing*, 39(2):546–563, 2009. 25

[8] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In *Proceedings 6th International Workshop on Algorithms and Data Structures (WADS)*, volume 1663 of *Lecture Notes in Computer Science*, pages 342–351. Springer, 1999. ISBN 3-540-66279-0. 44, 45, 51, 52, 53

[9] Cheng Cao. Study on two optimization problems: Line cover and maximum genus embedding. Master's thesis, Texas A&M University, 2012. 22, 25

[10] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 365–372. ACM, 1987. 49

[11] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer Publishing Company, Incorporated, 1st edition, 2012. ISBN 3642648738, 9783642648731. 31

[12] Herbert Edelsbrunner, Leonidas Guibas, and Micha Sharir. The complexity of many cells in arrangements of planes and related problems. *Discrete & Computational Geometry*, 5 (1):197–216, 1990. 32

[13] György Elekes and Csaba D Tóth. Incidences of not-too-degenerate hyperplanes. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 16–21. ACM, 2005. 22, 32

[14] David Eppstein and Emma S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. *Journal of Graph Algorithms and Applications*, 16(2):543–567, 2012. 44

[15] Vladimir Estivill-Castro, Apichat Heednacram, and Francis Suraweera. FPT-algorithms for minimum-bends tours. *International Journal of Computational Geometry & Applications*, 21(02):189–213, 2011. 22

[16] Jacob Fox, János Pach, Adam Sheffer, Andrew Suk, and Joshua Zahl. A semi-algebraic version of Zarankiewicz's problem. *arXiv preprint arXiv:1407.5705*, 2014. 22

[17] Magdalene Grantson and Christos Levcopoulos. *Covering a set of points with a minimum number of lines*. Springer, 2006. 22

[18] Ben Joseph Green and Terence Tao. On sets defining few ordinary lines. *Discrete & Computational Geometry*, 50(2):409–468, 2013. 22

[19] Leonidas J Guibas, Mark H Overmars, and Jean-Marc Robert. The exact fitting problem in higher dimensions. *Computational geometry*, 6(4):215–230, 1996. 22

[20] LJ Guibas, Mark Overmars, and Jean-Marc Robert. The exact fitting problem for points. In *Proc. 3rd Canadian Conference on Computational Geometry*, pages 171–174, 1991. 22

[21] Meng He, Ganggui Tang, and Norbert Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In *Proceedings 25th International Symposium on Algorithms and Computation (ISAAC)*, volume 8889 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 2014. 44, 45, 51, 52, 53

[22] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992. 44

[23] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In *Proceedings 41st International Colloquium Automata, Languages, and Programming (ICALP), Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 532–543. Springer, 2014. 44, 45, 51, 53

[24] Łukasz Kowalik. Adjacency queries in dynamic sparse graphs. *Information Processing Letters*, 102(5):191–195, 2007. 44, 45, 52

[25] Stefan Kratsch, Geevarghese Philip, and Saurabh Ray. Point line cover: The easy kernel is essentially tight. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1596–1606. SIAM, 2014. 25

[26] VS Anil Kumar, Sunil Arya, and Hariharan Ramesh. Hardness of set cover with intersection 1. In *Automata, Languages and Programming*, pages 624–635. Springer, 2000. 22, 24

[27] Stefan Langerman and Pat Morin. Covering things with things. *Discrete & Computational Geometry*, 33(4):717–729, 2005. 22, 24

[28] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26(3):269–277, 1988. 49

[29] Min Chih Lin, Francisco J. Soulignac, and Jayme L. Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *Theoretical Computer Science*, 426:75–90, 2012. 44

[30] Nimrod Megiddo and Arie Tamir. On the complexity of locating linear facilities in the plane. *Operations research letters*, 1(5):194–197, 1982. 22, 24

[31] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Transactions on Algorithms (TALG)*, 12(1):7, 2016. 17, 45

[32] János Pach and Micha Sharir. On the number of incidences between points and curves. *Combinatorics, Probability and Computing*, 7(01):121–127, 1998. 22, 27

[33] David Peleg and Shay Solomon. Dynamic $(1+\varepsilon)$-approximate matchings: a density-sensitive approach. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 712–729. Society for Industrial and Applied Mathematics, 2016. 45

[34] Shay Solomon. Fully dynamic maximal matching in constant update time. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 325–334. IEEE, 2016. 45

[35] József Solymosi and Terence Tao. An incidence theorem in higher dimensions. *Discrete & Computational Geometry*, 48(2):255–280, 2012. 22

[36] Endre Szemerédi and William T Trotter Jr. Extremal problems in discrete geometry. *Combinatorica*, 3(3-4):381–392, 1983. 22, 25

[37] Praveen Tiwari. On covering points with conics and strips in the plane. Master's thesis, Texas A&M University, 2012. 22

[38] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937. 3

[39] Jianxin Wang, Wenjun Li, and Jianer Chen. A parameterized algorithm for the hyperplane-cover problem. *Theoretical Computer Science*, 411(44):4005–4009, 2010. 10, 22

[40] Stephen Wolfram. Data science of the Facebook world, 2013. URL `http://bit.ly/1aYGsoF`. Accessed September 10, 2017. 13

[41] Mark Zuckerberg. Facebook post, 2017. URL `http://bit.ly/2sefWFL`. Accessed September 10, 2017. 13