Parameterized Relaxations for Circuits and Graphs

by

Shyan Akmal

B.S. Harvey Mudd College (2019)
S.M. Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Shyan Akmal Department of Electrical Engineering and Computer Science May 17, 2024
Certified by:	Virginia Vassilevska Williams Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Certified by:	Ryan Williams Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Parameterized Relaxations for Circuits and Graphs

by

Shyan Akmal

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

What makes some problems hard to solve, and others easy? In situations where complexity-theoretic hypotheses rule out the possibility of fast algorithms for problems, are there nonetheless instances for which we can evade intractability and still design efficient algorithms? In this thesis, we investigate these questions from the perspective of *parameterized relaxations*. We consider important computational problems on circuits and graphs, and design fast algorithms for relaxed versions of these tasks, that highlight tractable instances of problems which are provably hard in general.

On circuits, we tackle the Majority-SAT problem, a task related to counting solutions to Boolean formulas in conjunctive normal form (i.e., CNF formulas), which has been extensively studied in areas related to probabilistic planning and inference. It has been known since the problem's introduction in the 1970s that Majority-SAT is complete for the class PP(intuitively, the complexity class of decision versions of counting tasks, believed to contain very difficult problems), and so under standard conjectures in complexity theory, cannot be solved in polynomial-time. We nonetheless show however, that Majority-SAT can be solved in optimal linear time when its inputs are restricted to be k-CNF formulas (i.e., CNF formulas where every clause width at most k), for any constant integer $k \quad 1$. This is surprising, since most circuit satisfiability problems remain hard even when restricted to 3-CNF formulas. Indeed, prior to our work, it was widely conjectured that Majority-SAT should be PP-complete on 3-CNFs. Beyond overturning this conjecture, we also characterize the complexity of many additional variants of Majority-SAT on bounded width formulas.

On graphs, we tackle the All-Pairs Connectivity and Disjoint Shortest Paths problems.

In the All-Pairs Connectivity (APC) problem, we are given an unweighted, directed graph G on n vertices, and are tasked with computing the maximum flow between each pair of vertices in G. Despite significant research on the problem, the fastest algorithm for APC in dense directed graphs is the naive $n^{4+o(1)}$ time approach, which simply runs a fast maximum flow algorithm separately for each pair of nodes. Moreover, the Strong Exponential Time Hypothesis (SETH) implies that APC cannot be solved in truly subcubic time. We consider a relaxation of APC, the k-Bounded All-Pairs Connectivity (k-APC), problem for integer $k \quad 1$, where for each pair of nodes $(s; t)$ in G, we must compute the maximum flow from s to t exactly if the maximum flow value is less than k, but if the maximum flow is at least k we merely need to report that the flow value is "large" instead of computing its exact value. We present an algorithm solving k-APC in $O((kn)^{!})$ time, where $! < 2:3716$ is the exponent of matrix multiplication. This is subcubic even for small k (evading the SETH lower bound

for the general APC problem), and runs in $\tilde{O}(n^{\ell})$ time for all constant k, which is already optimal for the 1-APC problem under conjectures in fine-grained complexity. Before our work, no algorithm was even known for 3-APC that ran faster than an algorithm simply solving the general APC problem directly.

In the Disjoint Shortest Paths (DSP) problem, we are given a graph G on n vertices, with specified source nodes $s_1, \ldots, s_k$ and target nodes $t_1, \ldots, t_k$, and are tasked with determining if G contains internally vertex-disjoint s_i t_i shortest paths. This problem is NP-hard in general, if k can grow with n. We study k-DSP, the DSP problem parameterized by the number of terminal pairs, for small k. We show that 2-DSP can be solved in optimal linear time over weighted undirected graphs and directed acyclic graphs. Prior to our work, the fastest algorithm for 2-DSP over weighted undirected graphs took $O(n^7)$ time, and the fastest algorithm over weighted, dense directed acyclic graphs took $O(n^3)$ time.

Thesis supervisor: Virginia Vassilevska Williams
Title: Professor of Electrical Engineering and Computer Science

Thesis supervisor: Ryan Williams
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

> A Palleseen writer once described the world of superstition and unreason they were bringing perfection to as 'the Great Night', contrasted with the clarity of their rational sun. Which writer was subsequently excised from the Pal canon because personification of an abstract is in itself irrational. **Nonetheless, the night persists.**
>
> Adrian Tchaikovsky, *House of Open Wounds*

In this Great Night, there are many lights I am grateful to.

Naturally, I'd like to begin by thanking Virginia Vassilevska Williams and Ryan Williams for being phenomenal advisors. Each of them brings a poise, intellectual strength, and clarity of thought I find admirable and inspiring. They have both provided me with excellent guidance, engagement, and encouragement, and have helped me cultivate a deeper appreciation for many topics in theoretical computer science. There have been many times I felt unsure about whether I was cut out for research, but after a quick conversation with Virginia or Ryan I felt refreshed, and eager to tackle any problem that came my way.

I also thank Ronitt Rubinfeld for agreeing to be the third member of my thesis committee. Ronitt was one of the first faculty members I met at MIT, and since that time has consistently remained one of the friendliest people I've talked with in grad school. I also appreciate the many opportunities she gave me to practice lecturing while I was the teaching assistant in her sublinear-time algorithms course, and the help she provided with postdoctoral applications. Additionally, I thank Joanne Hanley, Ronitt's admin, for being very helpful in facilitating communication with Ronitt.

Rationally, I thank Ce Jin, Yinzhan Xu, Rahul Ilango, Nicole Wein, Surya Mathialagan, Zixuan Xu, Andrea Lincoln, Jenny Kaufmann, Lijie Chen, Caleb Robelle, Yael Kirkpatrick, Jiatu Li, and Ted Pyne for being excellent academic siblings. Despite some of their individually nonzero fraction parts, they have all been an integral part of my time at MIT.

Ce has been a phenomenal collaborator. I've greatly enjoyed the time I've spent thinking about problems with him, and hope we find chances to continue working together in the future. I also thank him for willing to join me for complicated board games. Yinzhan has shown me many cool graph theory ideas, distracted me with silly math problems on at least one long train ride, defeated me in Time Travel Chess and Beast many times in the past (and hopefully in the future), cooked me nice food, and has overall been an excellent friend these past five years. Rahul has been one of the friendliest faces in the theory group, and I thank him for making the office a warmer place. I greatly appreciate his consistency in asking people what the dependence of the parameter k is in their results. Nicole is one of the nicest

people I met at MIT. She was a major part of what made the theory group so welcoming when I first joined. I've learned a lot from collaborating with her, and admire how effectively and systematically she is able to extract meaningful structure from seemingly complicated graph theoretic objects. Nicole also introduced me to the game *Bean and Nothingness*, which I have had a lot of fun with—Bean Studies is no joke. Surya made the theory group a fun and funnier place. I appreciate her infectious sense of humor and willingness to share lots of cool art. I also thank her for being someone I could talk about Pokémon with. I thank Zixuan for fun collaborations, and for sharing my interest in algebraic techniques.

For algebraic reasons, I thank Sandeep Silwal, Justin Chen, Anthimos-Vardis Kandiros, Dhruv Rohatgi, and Noah Golowich. Sandeep is a person who acts as positive force in many people's lives. I thank him for always being willing to tell me about cool applications of polynomials, joining me for board games, sleeping through board games, consistently telling his audiences to not ask him any questions (and then clarifying that was supposed to be a joke), and overall shaping the theory group to be fun and welcoming community. Justin is a kind soul I feel lucky to have met in grad school. I thank him for many nice walks, pleasant conversations, experimental movie nights, and memorable board game victories and losses. I thank Vardis for joining me for lots of fun events throughout grad school, agreeing with me about food preferences, and reimbursing me before he fled to California. I thank Dhruv Rohatgi for showing me there exist MIT students with good taste in books. Getting to corner Dhruv and talk with him about science fiction and fantasy was a frequent highlight of theory group events for me. I thank Noah Golowich for nice conversations at MIT and, together with Rahul Ilango, for gifting me a matrix multiplication meme that one time.

For reasons of prime importance, I thank Abrar Haque, Hamzah Ahmed, Adeeb Chowdhury, Areeb Alam, Farhan Habib, and Zain Hannan, for many fun times over the past two decades. Abrar has been an incredibly supportive friend, who never fails to make me laugh. Hamzah has been an incredibly supportive friend, who frequently fails to make me laugh. Nonetheless, I appreciate the creative math memes and silly conversations he brings to the table. I thank Adeeb for hosting Thanksgivings at his place in New York, giving me lots of science fiction and film recommendations over the years, and making me laugh by calling me a fish tank. Areeb was one of my first friends, and having him nearby at Harvard made life in grad school much better. I thank Farhan for hosting a Thanksgiving at his home that was one of the best vacations I have had in the past five years, and being someone I can talk about cool fiction with. I thank Zain for being someone I can nerd out about board games with, and for being the best dungeon master I could ask for. I greatly appreciate the impact all these people have had on my life.

For transcendental reasons, I thank Karina Cho, Jordan Haack, Chi-Ning Chou, and Amir Abboud. Karina has been a great friend, and I appreciate her well wishes over the years in grad school. Much of my own values in mathematics and pedagogy have been shaped by talking with her and seeing her work. I also thank her for gifting me some art which cheers me up everyday. Jordan is someone I always appreciate getting to talk with math about. I thank him for the many fun times and fond memories. I thank Chi-Ning for being a welcoming presence when I stopped by the Harvard theory group. I thank Amir for being an approachable researcher during my first year as a grad student, and a friendly face in virtual conferences during the lockdown era. I feel fortunate to have talked with him so early in my grad school career, and look forward to working with him in the near future.

I'd really like to thank the MIT libraries staff for buying me forty books while I was a graduate student, and helping me procrastinate on research.

For complex reasons (but none purely imaginary), I thank Sakib Haque, Tejas Jayashankar, and Uma Ilavarasan. Sakib has been a great friend over the years, who has checked up on me when others would not think to. I appreciate his thoughtfulness and sense of humor, and look forward to many more delightful conversations in the years to come. Tejas has been a reliable source of joy in difficult times. One of the main downsides of leaving Massachusetts is that I will not be able to see him as frequently. I thank him for fun outings, board games, movies, meals, and advertising my fun fact about stop signs on day that one time. Uma has been a lively and positive presence in Massachusetts. I feel fortunate that, like Areeb, she joined Harvard nearby MIT. I thank her for fun feline photos, dramatic games of Betrayal, invigorating games of Spirit Island, and lots of good food.

For surreal reasons, I thank Dylan McKay for suggesting I watch Cowboy Bebop, Adam Busis for doing his best to prevent me from learning how to integrate, Arsen Vasilyan for hidden reasons, Manasi Vyas for liking my Goodreads updates, and Charles Leiserson for offering lots of meaningful work and life advice that I will take with me into the future (and also being a graduate counselor who let me do whatever I wanted).

For digital reasons, I thank Ce Jin for pointing out several mistakes in an earlier version of this thesis; Laura Brandt for offering extensive comments which helped improve the writing of this thesis; Ryan Williams for proofreading part of this thesis; and Sepehr Assadi and Jason Li for help with LaTeX.

For reasons of human flourishing, I thank Mohamed Omar, Jim Boerkoel, JJP Veerman, Ran Libeskind-Hadas, and Francis Su for being inspiring mentors.

With high probability, I'd almost surely like to thank Abhijit Mudigonda, Allen Liu, Lijie Chen, Laura Brandt, Kwangjun "KJ" Ahn, Nicholas Draper, Jingnan Shi, Tomohiro Koana, Vivaswat Ojha, Sujit Rao, Shyam Narayanan, and Aviad Rubinstein. I thank Abhijit for visiting me at MIT several times, attending my defense virtually, and for sharing fun computer science and puzzle links with me. I also appreciate him humoring my many complaints over the years. I thank Lijie for some nice meals and board games during grad school, and technical writing advice that I have found very helpful. I thank Laura for introducing me to many cats, fun board games, cool restaurants, interesting books, and being a good friend. I thank KJ for nice conversations and telling me about areas of computer science outside my specific area of research.

For satisfying reasons, I thank Quinten Tupker, Till Tantau, and Olaf Beyersdorff for a nice discussion at Dagstuhl Seminar 23111, which gave rise to Theorem 4.19 in this work.

For reflective reasons, I thank James B. Wilson, Abhijit Mudigonda, Andrea Lincoln, Nicole Wein, Ryan Williams, Ivan Mihajlin, Alexander S. Kulikov, and Ray Li for placing me in the acknowledgements sections of their works [DW22, MW21, Lin20, Wei21, Wil24, BGK+23, HL23, BW24].

For recursively enumerable reasons, I thank my extended family—grandparents, aunts, uncles, cousins, and beyond—for cheering me on through my PhD years.

For p-adic reasons, I thank Tahsin Saffat. Tahsin is the stone in the pond whose ripples brought me here. I only got interested in mathematics because I saw how fascinated Tahsin was with it. He has been one of the most supportive and insightful people in my life, and his intellectual curiosity continues to be an inspiration to me. My gratitude to him is

endless. Whether he is pontificating about the Langlands program, sharing a cool puzzle, insisting that "child's drawings" is an appropriate term for an embedding, moping because he drew Elemental Boon and Powerstorm in the same turn of Spirit Island, pondering the existential philosophy of Albert Camooose, expressing confusion, or earnestly explaining to me how *high-dimensional spaces really exist*, I cherish every moment I have had with him, and hope for many more. As we both venture into this Great Night, may we unveil many beautiful mysteries.

For reasons of cardinal importance, I thank my brothers, Zayan Akmal and Ryaan Akmal, who have been an immense source of support and entertainment over the past twenty-two years. Zayan contains multitudes, and is a great joy to talk with. I appreciate how he sends me wild math ideas he has, tells me about the nexus between physics and chemistry, conveys postmodern comedy that I cannot begin to appreciate, and shares the aspects of life that he enjoys. Ryaan contains, if not multitudes, then at least two tudes (which is more than enough for anyone). Conversations with him over Zoom have helped keep me sane in grad school. I appreciate his willingness to dive deep into themes across fiction, share what is going on in his life, listen compassionately, and take initiative to make the lives of those around him better. My life would be unrecognizable and impoverished without them, and I thank them both for all they have done.

For transfinitely many reasons, I thank my mother, Nahid Farhana, and my father, Sayeed Akmal. The biggest blessing in my life is having these two as parents. No matter how far I am in the world, my home is where my mother is. Throughout undergrad and grad school, my mom is the one who has most consistently called and messaged me, to check in on how I am doing. I am proud of her for having completed her own graduate degree faster than I completed mine, and thank her for helping me feel loved at all times. My dad has similarly provided me with a gargantuan level of support. He has always believed more in my abilities than I have, and I appreciate the confidence he has in me. In recent years, I have greatly enjoyed bonding with him over science fiction books and movies. I thank him and my mom for instilling me with positive values. The depth of my gratitude for all the moments we have spent together, big and small, remembered and forgotten, is a well that cannot run dry. I thank my parents for being the best role models I could ask for.

Contents

List of Figures

Chapter 1

Introduction

> . . . here I am . . . crammed with ideas, and visions, and so on, and can't dislodge them, for lack of the right rhythm. Now this is very profound, what rhythm is, and goes far deeper than words. A sight, an emotion, creates this wave in the mind, . . . and then, as it breaks and tumbles . . . it makes words to fit.
>
> Virginia Woolf, *The Letters of Virginia Woolf: Volume Three*

> The ground of our experience is dark, and all our inventions start in that darkness. From it, some of them leap forth in fire.
>
> Urusula K. Le Guin, *The Wave in the Mind*

> Let me plant a seed . . . and when you come back, we'll see what has grown.
>
> Mark Lawrence, *The Book That Wouldn't Burn*

Sometimes the problems we face feel impossible to solve. This is especially true in theoretical computer science, where *impossibility results* present a major obstacle to designing faster algorithms. Conjectures from complexity theory imply that, for many important computational tasks, existing (often slow or naive) algorithms are essentially optimal. Nonetheless, these tasks still need to be solved in practice, and as datasets continue to grow in size, faster algorithms become more and more necessary. In this thesis, we grapple with this intractability by studying *parameterized relaxations* for foundational computational problems on circuits and graphs. Studying relaxations of computational tasks is a way of going beyond worst-case analysis and *identifying tractable instances* of difficult problems. Besides gaining a better understanding of which problem instances admit efficient algorithms, this approach also leads to a deeper structural understanding of the discrete mathematical objects—circuits and graphs—our algorithms analyze.

What is a *relaxation*?

A ***relaxation*** of a computational problem is just an easier version of that task. For a relaxation to be meaningful, it should be a mathematically natural and interesting problem, and solving it should provide useful information, relevant to the goals of the original task.

In this thesis, we explore two ways of relaxing a computational problem.

In the first approach, we relax the possible ***inputs*** to a problem, by restricting the space of instances we are expected to handle. For example, instead of solving a circuit analysis task on arbitrary formulas from a certain class of circuits, we can relax the problem by restricting our attention to an interesting subclass of circuits.

In the second approach, we relax the quality of the information we are expected to ***output*** when solving a problem. For example, instead of computing the exact value of a metric, we may just return some information about the relative size of the metric.

What makes a relaxation *parameterized*?

A ***parameterized*** algorithm is one whose runtime depends on not just the input size n, but also on an auxiliary parameter k, that bounds the complexity of the problem we are considering in some way. In the context of relaxations, k can be viewed as a tuning parameter, which interpolates between easier problem variants (for small k) and the general, possibly intractable original problem (for large k). For example, k might correspond to a degree bound on the instances we encounter in the input relaxation of a problem, or might correspond to a threshold we need to compare the value of a metric to in an output relaxation.

In general, there is a rich, well-developed theory of ***parameterized complexity*** in computer science, which studies the runtime dependence of algorithms in both the parameter k and input size n (see e.g., [CFK+16, FLSZ18]).

Why are parameterized relaxations interesting?

In this thesis, we explore parameterized relaxations for problems that are computationally intractable under popular conjectures in complexity theory. We design fast algorithms for these relaxations. In doing so, we identify parameters of interest k related to the problem structure, and show that the original (in general, intractable) problems can be solved efficiently for small values of k.

Studying parameterized relaxations in this way lets us more precisely pinpoint the ***true source of intractability*** in these tasks, by tying the difficulty of the tasks to the value of k. This type of multivariate algorithm design provides a deeper mathematical insight into which instances of these problems are easier or harder. This is interesting both from the perspective of ***theory***, where we care about understanding what properties of a structure, such as a circuit or graph, make certain types of computation difficult or easy, and the viewpoint of ***practice***, where we want to know whether real-world instances of the problems we encounter can be solved faster, using instance-dependent structure.

Next, we present high-level descriptions of our results, and explain how they fit into the paradigm of parameterized relaxations.

1.1 Overview of Results

We design algorithms for problems on circuits and graphs.

Circuits

Many central questions in computer science involve analyzing the solution spaces of circuits. Completeness results in theoretical computer science have made understanding the complexity of circuit analysis problems a cornerstone problem in the field. For example, in the Satisfiability (SAT) problem, we are given a CNF formula—intuitively, a Boolean formula defined by a collection of constraints on the variables—and are tasked with determining if the formula is satisfied—evaluates to true—under some assignment to its variables. SAT is NP-complete, and so we can use reductions from SAT to show hardness for a plethora of important combinatorial problems.

Majority-SAT is a variant of SAT, where we are again given a CNF formula, but are now tasked with determining if *at least half* of all assignments to the variables of the formula satisfy it. Majority-SAT is closely tied to the problem of counting satisfying assignments, and is complete for the class PP, a complexity class which is believed to contain problems much harder than NP-complete problems. The intractability of Majority-SAT is very important in subareas of Artificial Intelligence related to logical inference and probabilistic planning, because reductions from Majority-SAT are the primary way of showing hardness for a variety of problem of interest which arise in these fields. Given an integer k 1, a k-CNF formula is CNF formula where each constraint contains at most k variables—intuitively, the formula is built out of small "local constraints."

Due to the centrality of Majority-SAT as a source of hardness in the fields mentioned above, understanding the complexity of Majority-SAT on k-CNFs for constant k is a research question which drew significant attention from various communities in computer science, since hardness for this task would enable researchers to show hardness for various "local" and "bounded-degree" variants of tasks in inference and planning. In this context, a large number of papers pointed out that determining the complexity of Majority-SAT on k-CNFs for k 2 $f2; 3g$ was open, with many conjecturing that the problem should be PP-hard, or in some cases erroneously asserting that the problem was known to be PP-hard: [Mun00a, Mun00b, BDK01, KG05, BDK07, GHM08, KG09, TF10, PLMZ11, Kwi11, FGL12, KdC15a, KdC15b, MDCC15, CDdB16, CM18, BDPR19, BDPR20]. We show, contrary to these conjectures, that Majority-SAT can be solved in linear time on k-CNF formulas, for any constant k.

Theorem: Solving Threshold Satisfaction on k-CNFs

For any fixed positive integer k and constant p 2 $(0; 1)$, there is a linear time algorithm which takes as input a k-CNF formula $'$ and determines if Pr[$'$] p. In particular, Majority-SAT can be solved on k-CNFs in linear time for any constant k.

The core idea behind our algorithm is a *regularity lemma* for bounded width formulas, which shows that for any constant k, a k-CNF formula with a large fraction of satisfying assignments can, after asserting a small number of constraints, be transformed into a formula

whose solution space is the disjoint union of the sets of satisfying assignments of 1-CNFs.

Beyond this result, we also consider various generalizations of Majority-SAT on bounded width formulas, and show how the complexity of these tasks can drastically change even under seemingly minor modifications to the problem definitions.

From the perspective of parameterized relaxations, we took the intractable problem Majority-SAT, relaxed the class of inputs it was over to k-CNF formulas, and achieved a problem we could solve efficiently for any constant k. This result gives us a clearer understanding of when and why the Majority-SAT problem is hard—the difficulty of the Majority-SAT problem comes from the presence of clauses of large width in its input formulas.

Graphs

Over graphs, we study problems which involve *detecting disjoint paths*.

Connectivity

Finding maximum flows is a central problem in computer science, with applications throughout graph algorithms, operations research, optimal transport, and combinatorial optimization. In unweighted graphs (i.e., graphs where edges have unit capacity), the maximum flow from s to t specializes to a classic measure known as the *connectivity* $(s;t)$ from s to t, defined to be the maximum number of edge-disjoint s t paths in G.

In the All-Pairs Connectivity (APC) problem, we are given an unweighted graph G, and are tasked with computing the connectivity $(s;t)$ for all pairs nodes $(s;t)$ in G. Solving APC is useful for the same reasons that finding maximum flow is useful. APC is relevant for example, when we want to identify to what extent different parts of a network are well-connected.

In dense, directed graphs, the current fastest algorithm for APC is the completely naive approach, which simply computes $(s;t)$ separately for each pair of nodes $(s;t)$ using a fast maximum flow algorithm. If G has n nodes and m edges, this algorithms solves APC in $n^2m^{1+o(1)}$ time, since maximum flows can be found in almost-linear time [CKL+22]. Moreover, conjectures in fine-grained complexity imply that solving APC requires n^3 $^{o(1)}$ time.

The lack of algorithmic progress for solving APC in general graphs has motivated researchers to study bounded variants of the problem. In this context, for integer k 1, we study the k-Bounded All-Pairs Connectivity (k-APC) problem, where we are given the same input as in APC, but now must merely return the value of $\min(k;\ (s;t))$ for each pair of nodes $(s;t)$ in G. Intuitively, k represents a cutoff distinguishing between low and high connectivities in G. In k-APC we are expected to compute the values of all low connectivities, but for pairs with high connectivity we just need to report that the connectivity value is large instead of computing it exactly. This is relevant in applications where a network is experiencing edge failures for example, and it is important to identify which pairs of nodes are in danger of losing their connections because of low connectivity, while for well-connected nodes knowing their precise connectivity value is not as important.

It has been known since the 1970s that 1-APC can be solved in $O(n^!)$ time [FM71], where $! < 2{:}3716$ is the exponent of matrix multiplication [WXXZ24]. Similarly, 2-APC can be solved in $O(n^!)$ time [GGI+17]. However, already for $k = 3$, it was unknown whether k-APC

could be solved faster than the general APC problem! We show that indeed it can, by solving k-APC for any k in $O((kn)^{\ell})$ time.

Theorem: Computing k-Bounded Connectivities

Given a positive integer k and a directed graph G on n vertices, we can compute $\min(k; \ (s;t))$ for all pairs of vertices $(s;t)$ in G in $O((kn)^{\ell})$ time.

For $k \quad n^{0:2}$, for example, our k-APC algorithm runs in subcubic time, thereby avoiding the $n^{3 \ o(1)}$ time lower bound for the general APC problem. For all constant k, our algorithms run essentially as quickly as the fastest known algorithm for 1-APC, which is optimal under certain hypotheses from fine-grained complexity.

Our algorithm is based off an algebraic framework, which enumerates families of edge-disjoint paths in G using determinants of suitably defined matrices. Beyond the result mentioned above, we illustrate the flexibility of our enumerative perspective, by showing how it implies fast algorithms for other bounded variants of APC, such as an all-pairs vertex connectivity and global minimum vertex-cut problem.

Disjoint Shortest Paths

For integers $k \quad 1$, in the k-Disjoint Paths (k-DP) problem, we are given a graph G with specified source nodes $s_1; : : : ; s_k$ and target nodes $t_1; : : : ; t_k$, and are tasked with determining if G contains vertex-disjoint $s_i \quad t_i$ paths. For general k, this problem is NP-hard, and thus unlikely to admit a polynomial-time algorithm. However, for constant k, the k-DP problem can be solved in almost-linear time [KPS24]. Research in algorithms for k-DP has been very influential in graph theory because of the connections between this problem and results around forbidden minors [RS95].

More recently, researchers have studied an optimization variant of k-DP, the k-Disjoint Shortest Paths (k-DSP) problem, where we are given the same input as in k-DP, but are now tasked with determining if G contains vertex-disjoint $s_i \quad t_i$ *shortest* paths. Like k-DP, k-DSP is NP-hard for general k, and so to design polynomial-time algorithms for k-DSP, researchers have focused on the case of constant k. Unlike k-DP however, where essentially optimal algorithms are known for all constant k, the exact polynomial-time complexity of k-DSP remains poorly understood. Suppose G has n vertices and m edges. For all $k \quad 3$, the current fastest algorithms for k-DSP on weighted, directed acyclic graphs (DAGs) run in $O(mn^{k \ 1})$ time [FHW80, BK17], and on unweighted, undirected graphs run in $n^{O(k \ k!)}$ time (with no known results for weighted, undirected graphs) [BNRZ21].

For $k = 2$, faster algorithms are known: 2-DSP can be solved over weighted undirected graphs in $O(n^7)$ time [Akh20], and unweighted undirected graphs in $O(mn)$ time [BNRZ21]. Nonetheless, these runtimes are still slower than what is known for 2-DP.

We close this gap, by showing that 2-DSP can be solved in optimal linear time over weighted DAGs and undirected graphs.

Theorem: Detecting 2 Disjoint Shortest Paths

We can solve the 2-Disjoint Shortest Paths problem in weighted DAGs and weighted undirected graphs in linear time.

As with our k-APC algorithm, our 2-DSP algorithms use algebraic techniques, and involve constructing polynomials which enumerate pairs of vertex disjoint shortest paths in G. Our arguments observe that the structure of such polynomials simplifies drastically when we work over fields of characteristic two.

Unifying Themes

The main unifying theme behind our results is that of taking problems that are intractable in some regime, and identifying parameterized relaxations of those tasks that can be solved efficiently. Majority-SAT admits no polynomial-time algorithm under the widely-believed hypothesis that P $\not=$ PP, but can be solved in optimal linear time on k-CNFs for constant k. Similarly, k-DSP admits no polynomial-time algorithm assuming P $\not=$ NP, but can be solved in optimal linear time for $k = 2$. APC admits no truly subcubic time algorithm under SETH, but its output relaxation k-APC can be solved in subcubic time for small k.

Another way to view the results is that rather than trying to design a faster algorithm for the original problem (which might not be possible under popular conjectures), we identify a natural relaxation which we can *solve optimally*, and then design algorithms for problems which *interpolate between the easy and hard cases*. For example, solving Majority-SAT on 1-CNFs is trivial, because we can count satisfying assignments for 1-CNFs in linear time. We showed that in fact this linear-time complexity continues to hold for Majority-SAT on k-CNFs for all constants k 1. It has been known for over fifty years that 1-APC can be solved in $O(n^{!})$ time, and that this is essentially optimal assuming plausible conjectures related to the complexity of certain matrix multiplication problems. We showed that in fact this $O(n^{!})$ runtime can be recovered for k-APC for all constants k 1. For k-DSP, we did not show an optimal algorithm for all constant k, but rather achieved the first optimal algorithm for some nontrivial k, namely $k = 2$.

Finally, one more common theme of our algorithms is that they unveil deeper structural insights on the structures they work over. Our Majority-SAT algorithm on k-CNFs yields a regularity lemma for k-CNFs (as mentioned previously), but also implies that the possible values of fractions of satisfying assignments of k-CNFs for constant k are highly constrained compared to the fractions achieved by general CNF formulas. It had been known previously that connectivities in graphs are encoded in the inverse of a certain edge-adjacency matrix [CLL13]. Our k-APC algorithm shows that k-bounded connectivities can be encoded using a *low-rank* variant of the edge-adjacency matrix, so that graphs with small connectivities admit a sort of algebraic compression that is not known to hold for high connectivities. Finally, our 2-DSP algorithms show that the enumerative structure of pairs of disjoint shortest paths in weighted undirected graphs and DAGs simplifies greatly when viewed modulo two.

These are all structural insights about circuits and graphs that would not necessarily be clear from attacking the original problems, but are natural byproducts of our algorithms for parameterized relaxations.

1.2 Organization

In Part I we discuss our results for the Majority-SAT problem on k-CNFs, and in Part II we present our algorithms for graph problems. Within Part II, in Chapter 6 we introduce a standard framework for algebraic graph algorithms, and then in Chapters 7 and 8 use this framework to design our algorithms for connectivity problems and disjoint shortest path problems respectively.

Bibliographic Notes

The results of Part I are primarily based off work by Ryan Williams and myself in [AW22]. Some of our notation has been inspired by [Tan22a]. The results of Chapter 7 are primarily based off [Akm24], and work by Ce Jin and myself in [AJ24]. In Section 7.3, we present algorithms for the k-Bounded All-Pairs Vertex Connectivity and k-Vertex Connectivity problems which appeared previously in [AJ24] and [CR94] respectively. We present alternate proofs of correctness for these algorithms, using the framework of [Akm24]. The results of Chapter 8 are based off work by Virginia Vassilevska Williams, Nicole Wein, and myself in [AWW24]. Throughout, the proofs of our results have been streamlined and simplified from their original expositions in [AW22, Akm24, AJ24, AWW24].

1.3 General Preliminaries

Basic Notation

Given a positive integer a, we let $[a] = f1;:::;ag$ denote the set of the first a consecutive positive integers. We abbreviate the base 2 logarithm as $\log = \log_2$. Given functions f and g, we write $f(n) = O(g(n))$ if $f(n)$ $g(n)\,\mathrm{poly}(\log n)$.

Deterministic versus Randomized Algorithms

Throughout Part I, by default algorithms are deterministic, unless otherwise stated. Throughout Part II, by default algorithms are randomized and correct with high probability.

Part I

Circuits

Chapter 2

Meeting Majority Satisfiability

In this chapter, we give a terse, informal introduction to the concepts needed to understand the Majority-SAT problem and questions concerning its complexity. In Section 2.1, we review basic complexity classes and satisfiability problems, including Majority-SAT. Then in Section 2.2, we consider the complexity of satisfiability problems on formulas of bounded width, and discuss why the complexity of Majority-SAT on formulas of constant width was an open problem for so long. In Section 2.3, we collect useful facts about formulas and satisfaction probabilities, for use in later chapters. We conclude in Section 2.4 with an overview of the topics covered by the remaining chapters in Part I.

For formal details on the definitions of complexity classes and reductions, and a more accessible primer on complexity in general, we refer the reader to [Wig19, Chapters 3 to 4], and for more details on the classes #P and PP we refer the reader to [AB09, Chapter 17].

2.1 Complexity Classes and Complete Problems

What problems are easy to solve, and which are hard? Historically, computer science has attempted to provide concrete answers to these questions, by grouping problems into equivalence classes based on their difficulty, known as *complexity classes*. For example, the complexity class P (Polynomial Time) consists of all decision problems (i.e., problems with YES or NO answers) which can be solved in deterministic polynomial time. Intuitively, we think of P as a class of "easy problems."

A major goal of complexity theory is to separate P from the complexity class NP (Nondeterministic Polynomial Time). Intuitively, NP is the class of "problems with easily checkable answers." A decision problem is in NP if it can be solved by a deterministic polynomial-time verifier. A verifier is an algorithm which takes as input both the problem instance and a *certificate*, whose length is at most polynomial in the size of the problem instance, and then returns YES or NO. The verifier correctly solves a problem if, on instances where the correct answer is YES, there exists some certificate which makes the verifier return YES, and on instances where the correct answer is NO, the verifier returns NO regardless of the certificate.

For example, consider the Hamiltonian Path problem, where we are given a graph G, and are tasked with determining if G contains a simple path using all of its vertices. The Hamiltonian Path problem is in NP, because this problem is solved by the linear-time verifier

which checks if the certificate is a simple path passing through all vertices of G. In contrast, it is not known if Hamiltonian Path is in P, because even though verifying that a *given* certificate path uses all vertices of G is easy, it is not clear how to *find* such a path without using exponential time.

Complexity classes C can be analyzed by studying their *complete* problems—individual problems whose polynomial-time complexity completely determines whether C is contained in P or not. Many complexity classes of interest turn out to admit canonical complete problems which involve the analyzing the solution spaces of circuits. To define these problems, it will be helpful to first introduce a simple class of circuits, known as formulas in conjunctive normal form (or CNF formulas).

Definition 2.1 (CNF Formulas). Fix a collection of variables $x = (x_1, \ldots, x_n)$. A *literal* ℓ is a variable or its negation. In other words,

$$\ell \in \{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}.$$

A *clause* C over the variables in x is a disjunction

$$C = (\ell_1 \vee \cdots \vee \ell_r)$$

of *distinct* literals ℓ_j, no two corresponding to the same variable. We identify C with the set of literals $\{\ell_1, \ldots, \ell_r\}$ it contains. The *width* of C is defined to be the number $r = |C|$ of literals it contains. A clause of width w is often referred to as a w-clause. A 1-clause is also called a *unit* clause.

A formula φ in *conjunctive normal form*, or CNF formula (or just "CNF"), is a conjunction

$$\varphi = C_1 \wedge \cdots \wedge C_m$$

of distinct clauses. We identify φ with the set of clauses $\{C_1, \ldots, C_m\}$ it contains. If every clause in φ has width at most k, we say φ is a k-CNF formula (or just "k-CNF"). In this case, we also say φ has width k. The *size* of a CNF formula is defined to be the sum

$$|\varphi| = \sum_{i=1}^{m} |C_i|$$

of the sizes of the clauses it contains. A CNF formula φ is naturally viewed as a function

$$\varphi : \{0,1\}^n \to \{0,1\}$$

by setting $\varphi(a) = 1$ if and only if for every clause C in φ, the assignment $a \in \{0,1\}^n$ sets some literal of C to be true (where we identify 1 with "true" and 0 with "false").

We say a CNF φ is *equivalent* to a function $f : \{0,1\}^n \to \{0,1\}$ if $\varphi(a) = f(a)$ for all inputs $a \in \{0,1\}^n$. CNF formulas are expressive enough to model any Boolean function.

Proposition 2.2. For any function $f : \{0,1\}^n \to \{0,1\}$, there is a CNF φ equivalent to f.

Proof. For every assignment $a \in \{0,1\}^n$ such that $f(a) = 0$, consider the clause C_a which contains the literal $\neg x_i$ if $a_i = 1$, and contains the literal x_i if $a_i = 0$, for each $i \in [n]$. By construction, the clause C_a is satisfied precisely by the assignments from $\{0,1\}^n \setminus \{a\}$.

Consequently, if we define the CNF formula

$$\varphi' = \bigwedge_{\substack{a \in \{0,1\}^n \\ f(a)=0}} C_a$$

we see that φ' is satisfied by an assignment a if and only if $f(a) = 0$, so φ' is equivalent to f as desired.

Note however, that given a Boolean function f, the smallest CNF φ' equivalent to f might have size exponential in n, so not all functions can be represented by small CNF formulas. Intuitively, a CNF formula φ' is a succinct description of a Boolean function in terms of constraints. Each clause C is a constraint mandating that one of its literals be set to true, and φ' is true if and only if every constraint is satisfied.

We next introduce terminology for discussing the solution spaces of CNFs.

Definition 2.3 (Satisfaction Probability)**.** Given a function $f : \{0,1\}^n \to \{0,1\}$, we say that an input $a \in \{0,1\}^n$ is a *solution* or *satisfying assignment* of f, if $f(a) = 1$. In this case, we say the assignment a *satisfies* the function f. Given a function $f : \{0,1\}^n \to \{0,1\}$, we define

$$\Pr[f] = \frac{|\{x \mid f(x) = 1\}|}{2^n}$$

to be the fraction of assignments which satisfy f. Equivalently, $\Pr[f]$ is the probability a uniform random assignment satisfies f. We refer to the quantity $\Pr[f]$ both as the *fraction of satisfying assignments* and the *satisfaction probability* of f.

Since CNFs can be viewed as functions, Definition 2.3 extends to defining the satisfaction probability $\Pr[\varphi']$ for CNF formulas φ'. Given a CNF φ', one of the most basic questions we can ask about it is: does φ' have a satisfying assignment? This is the Satisfiability (SAT) problem, defined below in terms of satisfaction probabilities.

> Satisfiability (SAT)
>
> Given a CNF formula φ', determine if $\Pr[\varphi'] > 0$.

SAT is in NP, because it is solved by the linear-time verifier which checks if the input certificate is a satisfying assignment for the input formula φ'. Satisfiability is the cornerstone problem of complexity theory because it is the canonical complete problem for the class NP, meaning that not only is it in NP, it is in some sense *universal* for NP.

We next recall definitions related to completeness.

Definition 2.4 (Many-One Reductions)**.** A reduction from a decision problem A to a decision problem H is an algorithm which, given an instance I of A produces an instance I' of H, such that the answer to I' is YES if and only if the answer to I is YES.

A reduction from a problem A to a problem H presents a way of solving A if we have a way of solving H. If the reduction runs in polynomial time, then we know that if H can be solved in polynomial time, so can A, by chaining the polynomial-time reduction with the polynomial-time algorithm for H.

Definition 2.5 (Complete Problems). A problem H is *hard* for a complexity class C, if for *every problem* A 2 C, there is a polynomial-time reduction from A to H. We say H is *complete* for the class C if H 2 C and H is C-hard.

Given a complexity class C and problem H that is C-complete, if H has a polynomial-time algorithm, then because H is C-hard, *every problem* in C can be solved in polynomial time. Conversely, if H is not solvable in polynomial time, then H is an example of a problem in C that cannot be solved in polynomial time. So if a complexity class C has a complete problem H, the problem of determining whether C P (i.e., whether "all problems in C are easy to solve") corresponds precisely to determining whether the individual problem H has a polynomial-time algorithm. This latter task seems more approachable, since it involves working with one concrete problem, rather than an entire class of problems.

Proposition 2.6 (Cook-Levin Theorem). SAT is NP-complete.

See [Sip12, Theorem 7.30] for example, for a proof of Proposition 2.6.

One of the most important open problems in all of mathematics is the P versus NP question—are the complexity classes P and NP equal, or not? It is widely conjectured that in fact P $\not=$ NP, because if P = NP this would intuitively mean that finding a solution to a problem is no harder than recognizing that a proposed solution is correct, and this would defy our understanding of computation built over decades of research in algorithm design.

Given a problem H, if there is a polynomial-time reduction from SAT to H, then because SAT is NP-hard, H would also be NP-hard.

Under the hypothesis that P $\not=$ NP, no NP-hard problem can be solved in polynomial time. This enables us to use reductions from SAT to show conditional intractability for other problems. Indeed, a plethora of important problems in computer science are now known to be NP-complete via reductions from SAT (see e.g., [GJ79]), and assuming P $\not=$ NP, none of these problems can be solved in polynomial time.

So SAT has been used as a *source of hardness* for many problems in NP.

Not all problems of interest however, belong to the class NP. To show hardness for such problems, researchers have defined other complexity classes beyond NP, and identified variants of SAT which are analogously complete for these classes. In this thesis, two such classes will be important for us: #P ("Sharp P") and PP ("Probabilistic Polynomial Time").

Intuitively, #P is the class of "counting problems," where we are trying to count objects meeting certain conditions, and for any given object, we can "easily check" if it meets the relevant conditions. A function problem P (i.e., a problem where the answer is an integer) is in #P if there exists a deterministic polynomial-time verifier, such that for any instance of P, the number of certificates that make the verifier return YES on that instance is equal to the correct answer.

For example, consider the Satisfiability Counting (#SAT) problem, where we are given a CNF formula $'$, and are tasked with counting the number of satisfying assignments of $'$.

This task is in #P, by considering the linear-time verifier which returns YES if and only if the certificate is a satisfying assignment for ϕ (i.e., the same verifier which showed that SAT is in NP). Since the satisfaction probability of a CNF formula ϕ on n variables is just its number of satisfying assignment divided by 2^n, #SAT can be equivalently defined as the following problem:

Satisfiability Counting (#SAT)

Given a CNF formula ϕ, compute the value of $\Pr[\phi]$.

Just like how SAT is NP-complete, #SAT is #P-complete (under a natural generalization of the notion of reduction from Definition 2.4 to function problems). So under the hypothesis that not all problems in #P can be solved in polynomial time (a weaker assumption than P $\neq$ NP), we can use reductions from #SAT to show that various *counting problems* are #P-hard, and thus unlikely to be solvable in polynomial time either.

One technical difference between NP and #P is that the former is a class of *decision* problems, whereas the latter is a class of *function* problems. This makes directly comparing NP and #P awkward, because the types of problems these classes contain are different.

This motivates the complexity class PP, which is intuitively the "decision version of #P." A decision problem P is in PP if there exists a deterministic polynomial-time verifier, and polynomial-time computable function $c(\cdot)$ that assigns each instance I of P a positive integer certificate size $c(I)$ $\leq$ poly($|I|$), with the following behavior: for any instance I of P, if the correct answer to I is YES, then at least half of all certificates in $\{0,1\}^{c(I)}$ make the verifier return YES on that instance, and if the correct answer to I is NO, then at least half of all certificates in $\{0,1\}^{c(I)}$ make the verifier return NO on that instance.

For example, consider the Majority-SAT problem, where we are given a CNF ϕ, and are tasked with determining if ϕ is satisfied by at least half of all possible assignments to its variables. The same linear-time verifier which showed that SAT $\in$ NP and #SAT $\in$ #P implies (if we set the certificate size $c(\phi)$ to be the number of variables in ϕ) that Majority-SAT $\in$ PP.

Majority-SAT

Given a CNF formula ϕ, determine if $\Pr[\phi]$ $\geq$ 1=2.

Just like how SAT is NP-complete and #SAT is #P-complete, Majority-SAT is PP-complete, and this has been known since the problem's conception nearly fifty years ago [Gil74, Sim75]. So under the hypothesis that not P $\neq$ PP (an assumption that turns out to be weaker than P $\neq$ NP, as we discuss later), reductions from Majority-SAT can show hardness for exact threshold variants of counting problems.

The Significance and Intractability of Majority-SAT

As the canonical complete problem for PP, Majority-SAT acts as a central source of hardness for problems in PP. This is particularly relevant when studying problems related to probabilistic planning, scheduling, and inference, because PP and larger classes derived from it, as opposed to NP, appear to be the natural setting for analyzing the complexity of these tasks.

Knowing that a problem is PP-hard as opposed to just NP-hard can be viewed as stronger evidence of its intractability, because of results in complexity theory that suggest PP contains problems that are strictly harder to solve than problems in NP. To mention some of these results, it will be helpful to have the concept of a Turing reduction.

Definition 2.7 (Turing Reductions)**.** A Turing reduction from a problem A to a problem H is an algorithm which, given an instance I of A produces several instances $I'_1, \ldots, I'_t$ of H, and given the solutions to $I'_1, \ldots, I'_t$, produces the solution to I.

Definition 2.8. A problem H is *hard under Turing reductions* for a complexity class C, if for *every problem* $A \in C$, there is a polynomial-time Turing reduction from A to H.

If a problem H is hard for a complexity class C under Turing reductions, then just like in the case of the reductions defined in Definition 2.4, any polynomial-time algorithm for H can be used to solve all problems in C in polynomial time.

One result giving evidence that PP-complete problems might be inherently more difficult than NP-complete problems is Toda's theorem [Tod91], which implies that Majority-SAT is hard under Turing reductions for a complexity class PH ("Polynomial Hierarchy"), a vast strengthening of NP. It is conjectured that SAT is not hard under Turing reductions for PH (often referred to as the hypothesis that "the polynomial hierarchy does not collapse").

Distinguishing between the NP-completeness and the PP-completeness of problems is also useful understanding how hard various logic problems related to model counting are to solve in the real-world. This is because even though we do not have polynomial-time algorithms which solve NP-hard problems in the worst-case, in practice many real-world instances of NP-hard problems can be solved quickly in using heuristic SAT solver systems. In contrast, solving PP-hard problems requires programs for "model counting," which do not appear to be as successful as SAT solvers in practice. In contexts like these, which show up in probabilistic planning and scheduling for example, distinguishing between "easy" and "hard" problems is more of an NP-complete versus PP-complete question, rather than a P versus NP-complete question (see e.g., the discussion in the introduction of the talk [Dar21]).

Another related reason that PP-complete problems seem harder than NP-complete problems, is that the former can be used to solve counting problems. Given a CNF $'$, solving #SAT on $'$ gives us the value of $\Pr[']$, which we can then use to solve SAT and Majority-SAT on $'$ as well. In the other direction, it is not clear if we can efficiently reduce #SAT to SAT. However, we can efficiently reduce #SAT to solving a small number of instances of Majority-SAT. This follows from the fact that an algorithm for Majority-SAT can be modified to let us test if $\Pr[']$ p for *any* fixed threshold $p \in (0;1)$, not just $p = 1/2$.

Proposition 2.9. There is an algorithm that, given positive integers n and a with $a < 2^n$, constructs in $\mathrm{poly}(n)$ time a CNF formula over the variables $x_1, \ldots, x_n$ with $\Pr[\ ] = a/2^n$.

Proof. We describe an algorithm $\mathrm{Form}_n(a)$ with the desired behavior.

If $n = 1$, then $a < 2^n$ forces $a = 1$, and $\mathrm{Form}_1(1)$ returns the formula consisting of the single unit-clause $\{x_1\}$. We have $\Pr[\ ] = 1/2 = a/2^n$ so this has the desired behavior.

Suppose instead n 2. We perform casework based off the value of a.

If $a = 2^n$ 1, then we return $= x_1$ which has $\Pr[\ ] = 1/2 = a/2^n$ as desired.

If instead $a > 2^{n-1}$, we can write $a = 2^{n-1} + b$ for some positive integer $b < 2^{n-1}$, since by assumption we have $a < 2^n$. Recursively call $\mathsf{Form}_{n-1}(b)$ to output a formula φ' that does not use the variable x_n, such that $\Pr[\varphi'] = b/2^{n-1}$.

Construct the formula

$$\psi = \{C \cup \{x_n\} \mid C \in \varphi'\}$$

by adding x_n to every clause of φ'. We can construct ψ in linear time given φ'. An assignment satisfies ψ precisely when it either sets x_n to be true, or sets x_n to be false and the assignment to the remaining variables satisfies φ'. Thus returning

$$\psi = \Pr[x_n] + \Pr[\overline{x_n} \wedge \varphi'] = (1/2) + (1/2)\,(b/2^{n-1}) = (2^{n-1} + b)/(2^n) = a/2^n$$

has the desired behavior.

Otherwise, $a < 2^{n-1}$. In this case, recursively call $\mathsf{Form}_{n-1}(a)$ to output a formula φ' that does not use the variable x_n, such that $\Pr[\varphi'] = a/2^{n-1}$. Then return the $\psi = x_n \wedge \varphi'$. Since x_n does not appear in φ', we have

$$\Pr[\psi] = \Pr[x_n]\,\Pr[\varphi'] = (1/2)\,(a/2^{n-1}) = a/2^n$$

as desired.

Since we compute $\mathsf{Form}_n(a)$ by calling $\mathsf{Form}_{n-1}(b)$ on some integer b and spending at most linear time additionally, an easy induction argument shows that computing $\mathsf{Form}_n(a)$ takes at most $O(n^2)$ time, which proves the claim.

Corollary 2.10 (Majority-SAT is Threshold Independent)**.** For any real $p \in (0,1)$, the problem of deciding if a CNF formula φ on n variables satisfies $\Pr[\varphi] \geq p$ can be reduced in polynomial-time to solving Majority-SAT on a formula with $n+1$ variables.

Proof. Since φ has n variables, $2^n \cdot \Pr[\varphi]$ is a nonnegative integer less than or equal to 2^n. So without loss of generality, we may assume that $p = b/2^n$ for some positive integer $b < 2^n$.

Apply Proposition 2.9 with $a = (2^n - b)$ to construct, in $\mathsf{poly}(n)$ time, a CNF ψ on the same variable set as φ with $\Pr[\psi] = (2^n - b)/2^n$.

Let y be a variable not in φ or ψ. Construct the CNF formulas

$$\varphi' = \{C \cup \{y\} \mid C \in \varphi\}$$

and

$$\psi' = \{C \cup \{\neg y\} \mid C \in \psi\}$$

by adding y to each clause of φ and $\neg y$ to each clause of ψ respectively.

Then construct the CNF formula

$$\chi = \varphi' \wedge \psi'.$$

Given φ and ψ, we can construct χ in linear time.

An assignment satisfies χ precisely when either y is set true and the assignment to the remaining variables satisfies φ, or y is set false and the assignment to the remaining variables satisfies ψ. Thus

$$\Pr[\chi] = \Pr[y \wedge \varphi] + \Pr[\neg y \wedge \psi] = (1/2)\,(\Pr[\varphi] + \Pr[\psi]).$$

Since $\Pr[\ ] = (2^n \quad b)/2^n$, this implies that

$$\Pr[\ ] = (1/2) + (1/2)\ (\Pr[\ '] \quad (b/2^n)) :$$

The above equation implies that $\Pr[\ ] \quad 1/2$ if and only if

$$\Pr[\ '] \quad b/2^n = p:$$

Thus the map sending $'$ to $$ is a polynomial-time reduction proving the claim.

By Corollary 2.10, given a CNF $'$ on n variables and a real $p \in (0; 1)$, we can use a single call to an algorithm solving Majority-SAT on formulas with $n + 1$ variables and $\mathrm{poly}(n; |\ '|)$ size to check if $\Pr[\ '] \quad p$. Since $\Pr[\ '] = a/2^n$ for some nonnegative integer $a \quad 2^n$, we can combine the observation from the previous sentence with a binary search argument to compute $\Pr[\ ']$ *exactly* (i.e., solve #SAT) using $n+1$ calls to an algorithm solving Majority-SAT and polynomial-time overhead.

This justifies our intuition that PP is somehow a "decision version of #P" and shows that Majority-SAT is a flexible problem, with the specific choice of threshold $1/2$ in the problem's defining inequality $\Pr[\ '] \quad 1/2$ not mattering so much.

2.2 Formulas of Bounded Width

In Definition 2.1 we defined k-CNF formulas $'$ for any integer $k \quad 1$ as formulas where every clause has width at most k. Such bounded-width formulas arise naturally in applications which use CNF formulas to model problems determined by sets of "local constraints."

For each of our satisfiability variants SAT, #SAT, and Majority-SAT, one can ask whether the problem remains hard when restricted to k-CNFs for constant k. All these problems are easy when $k = 1$, because we can compute satisfaction probabilities of 1-CNFs in linear time.

Proposition 2.11 (1-CNF Satisfaction Probabilities)**.** There is an algorithm which takes as input a 1-CNF formula $'$, and computes $\Pr[\ ']$ in linear time. Moreover, $\Pr[\ ']$ is either zero or a power of two.

Proof. The algorithm scans through the clauses of $'$, each of which has a single literal.

If $'$ contains both x and $:x$ for some variable x, we return $\Pr[\ '] = 0$, since both of the clauses cannot simultaneously be true.

Otherwise, $'$ contains clauses coming from different variables. Let v be the number of distinct variables appearing in $'$. In a uniform random assignment, each clause containing a distinct variable of v is satisfied independently with probability $1/2$. So in this case we return $\Pr[\ '] = (1/2)^v$.

Thus we can compute $\Pr[\ ']$ in linear time, and the two cases above show that $\Pr[\ ']$ is either zero or a power of two as claimed.

So all the satisfiability problems we have mentioned, when restricted to k-CNFs for $k = 1$, belong to P. The complexity of these problems becomes more interesting when $k \quad 2$.

Reducing Width for SAT and #SAT

For any fixed integer $k \geq 1$, let kSAT be the SAT problem restricted to k-CNF formulas. Similarly, let #kSAT be the #SAT problem restricted to k-CNF formulas.

2SAT can be solved in linear time (e.g., by using the reduction from [Pap93, Section 9.2] to relate 2SAT to a a graph theoretic problem, and then running a linear-time algorithm for finding the strongly connected components of a graph), and this fact has been known in the literature since at least the 1970s [APT79].

Proposition 2.12 (2-CNF Satisfaction)**.** There is an algorithm that takes as input a 2-CNF formula $'$, and determines if $\Pr[\,'\,] > 0$ in linear time.

In contrast to 2SAT being easy, it turns out that #2SAT is already #P-hard.

Proposition 2.13. #2SAT is #P-hard under Turing reductions.

Proposition 2.13 was first proved in 1979, by [Val79a].

Once we jump up to $k = 3$, kSAT becomes as hard as the general SAT problem. This fact has been known for essentially as long as the NP-completeness of SAT has been known.

Proposition 2.14. 3SAT is NP-complete.

Proof. 3SAT is in NP because it is a special case of SAT, which is in NP.

We exhibit a polynomial-time reduction from SAT to 3SAT. This will prove the desired result, since SAT is NP-hard.

Take an arbitrary instance of SAT, consisting of a CNF formula $'$.

The idea of the reduction is to introduce new variables, which can be used to split each long clause of $'$ into smaller clauses of width 3.

For each clause C of width $w > 3$ in $'$, we introduce $(w - 1)$ new variables

$$y_{C;1}, \ldots, y_{C;w-1}.$$

If C is of the form

$$C = (\ell_1 \vee \cdots \vee \ell_w)$$

for some literals ℓ_i, we let $_C$ be the 3-CNF with clauses $(\ell_1 \vee y_{C;1})$,

$$(\neg y_{C;j} \vee \ell_{j+1} \vee y_{C;j+1})$$

for each $j \in [w - 2]$, and $(\neg y_{C;w-1} \vee \ell_w)$.

If instead C has width at most three in $'$, we define $_C = C$.

Now construct the 3-CNF formula

$$= \bigwedge_{C \in '} {}_C.$$

Given $'$, we can construct $$ in polynomial time.

We claim that $$ is satisfiable if and only if $'$ is satisfiable.

Indeed, suppose that $'$ is satisfied by some assignment a. This assignment must satisfy a literal in every clause C of $'$. For each such clause C, consider the subformula $_C$ of . If C has width at most three, $_C = C$ is satisfied by the assignment.

Otherwise, $C = (`_1 \_ \_ `_w)$ has width $w > 3$. Let $i \in [w]$ be an index such that a sets $`_i$ true. Then setting $y_{C;j}$ to be true for all $j < i$ and false for all $j \geq i$, together with a, satisfies $_C$ by construction. Setting the $y_{C;j}$ variables in this fashion for each clause C of width greater than three in $'$, we see that we can extend a to a satisfying assignment of , so is satisfiable.

Conversely, suppose is satisfied by some assignment a. Then a satisfies every clause C of width at most three in $'$ (since these clauses also appear in). Take any clause $C = (`_1 \_ \_ `_w)$ in $'$ of width $w > 3$. Since a satisfies , it satisfies $_C$. If a sets $y_{C;j}$ to be true for all $j \in [w - 1]$, then since a satisfies the clause $(:y_{C;w-1} \_ `_w)$, this assignment must set $`_w$ to be true. If instead a sets some $y_{C;j}$ variable to be false, let $i \in [w - 1]$ be the smallest index so that $y_{C;i}$ is set false by a. Then $_C$ has a clause containing the literals $y_{C;i}$ and $`_i$, and the literal $:y_{C;i-1}$ too if $i \geq 2$. By our choice of $i$, we know that $y_{C;i}$ is false, and $:y_{C;i-1}$ is false if $i \geq 2$. So in order for $a$ to satisfy $_C$, the assignment $a$ must set $`_i$ to be true. Since this holds for very clause C of width greater than three $'$, we see that the assignment a restricted to the variables of $'$ satisfies every clause of $'$.

Thus is satisfiable if and only if $'$ is satisfiable, which proves the desired result.

Barriers to Reducing Width for Majority-SAT

So far we have seen that #kSAT is #P-hard for $k \geq 2$, and kSAT is NP-hard for $k \geq 3$. Beyond presenting a nice complexity classification for variants of SAT and #SAT on bounded width formulas, these hardness results are important because they enable researchers to show hardness for *bounded degree variants* of important computational tasks.

So what about Majority-SAT? What is the complexity of this problem when we restrict to k-CNF formulas, for constant k? By analogy with the results for SAT and #SAT, it seems plausible that Majority-SAT on k-CNFs should be as hard as the general Majority-SAT problem for some small constant k, such as $k \in \{2; 3\}$.

As discussed in Section 1.1, determining the complexity of Majority-SAT on k-CNFs for *any constant* $k \geq 3$ was a major open problem, with it being widely conjectured that the problem should be PP-complete. We show that, in contrast to the intuition suggested by the behavior of SAT and #SAT on bounded width formulas, this conjecture is wrong, and in fact Majority-SAT on k-CNFs is in P for *any constant* $k \geq 1$. To state our result formally, we introduce the kSAT-Prob$_p$ problem, which restricts Majority-SAT to k-CNF formulas, and replaces the threshold of 1=2 with $p \in (0; 1)$.

> kSAT-Prob$_p$
>
> Given a k-CNF formula $'$, determine if Pr[$'$] $\geq p$.

We prove that kSAT-Prob$_p$ can be solved in linear time for constant $k \geq 1$ and $p \in (0; 1)$:

Theorem 2.15: Threshold Satisfaction for k-CNFs

For any fixed positive integer k and constant $p \in (0,1)$, there is a linear time algorithm solving kSAT-Prob$_p$.

Why should Theorem 2.15 be possible? For example, why can we not adapt the reduction from SAT to 3SAT in the proof of Proposition 2.14 to reduce 3SAT-Prob$_{1/2}$ to Majority-SAT, and thereby prove that 3SAT-Prob$_{1/2}$ is PP-complete?

Well, one issue is that the reduction as written does not preserve the count of satisfying assignments. Even if the reduction did take an arbitrary CNF formula $'$, and in polynomial-time construct a 3-CNF formula such that $'$ and have the same total number of satisfying assignments, the general reduction technique we use *increases the number of variables* by a large amount. Since the number of variables increases, the *fraction* of satisfying assignments in will be much smaller than the corresponding fraction for $'$, even though they have the same *number* of satisfying assignments. If and $'$ have the same number of satisfying assignments, but has v more variables than $'$, then $\Pr[\ ] = (1/2)^v \Pr[']$ will be vanishingly small, even for mildly superconstant v. So intuitively solving Majority-SAT on should not recover information about $\Pr[']$, because $\Pr[\ ]$ will be less than $1/2$ even if we only added two new variables in the reduction.

However, we saw in Corollary 2.10 that we can use an algorithm for Majority-SAT to determine if $\Pr[']$ p for a given CNF $'$ and any $p \in (0,1)$. Can this help us in designing a reduction from Majority-SAT to 3SAT-Prob$_{1/2}$? It cannot, because Corollary 2.10 crucially uses the assumption that Majority-SAT can be solved on formulas of unbounded width. In particular, the proof of Corollary 2.10 uses Proposition 2.9 to construct a CNF formula with $\Pr[\ ] = a/2^n$ for some integer a depending on p. However, for any constant k 1, there exist integers a such that *no k-CNF formula has satisfaction probability equal to $a/2^n$*. In general, if $'$ is a k-CNF, then $\Pr['] \in \left((2^k - 1)/2^k, 1\right)$. This is because if a k-CNF $'$ has $\Pr['] < 1$, it must contain a clause C of width at most k, so $\Pr['] \le 1 - \Pr[C] \le 1 - (1/2)^k$. So it is not clear that for fixed k, that having an algorithm for kSAT-Prob$_{1/2}$ would help us also solve kSAT-Prob$_p$ for $p \neq 1/2$.

This is a very concrete way in which k-CNF formulas, for constant k, are *less expressive* than general CNF formulas—the possible satisfaction probabilities they can achieve are more constrained. This might give some intuition for why Theorem 2.15 should be true.

> **Idea 1** Although for arbitrary CNF formulas $'$, $\Pr[']$ can be any value of the form $a/2^n$ for integers a and n, if $'$ is restricted to be a k-CNF for constant k, then the possible values $\Pr[']$ can take on are much more limited. In particular, there are intervals of constant size in $(0,1)$ not containing satisfaction probabilities of *any* k-CNF formula. We should be able to algorithmically exploit these gaps in satisfaction probablities of k-CNFs.

2.3 Helpful Facts

In this section, we collect additional helpful facts and constructions for working with CNF formulas and bounding their fractions of satisfying assignments.

Given a CNF formula $'$, a subformula $\quad '$ is a CNF whose clauses are all clauses of $'$. By default we view the subformula $\quad$ as having the same underlying variable set as $'$.

Proposition 2.16 (Subformula Satisfaction Probabilities)**.** Let $'$ be a CNF. Let $\quad '$ be a subformula of $'$. Then $\Pr[\,']\quad\Pr[\ ]$.

Proof. Since $\quad '$, every satisfying assignment of $'$ is also a solution for $\quad$. Since $'$ and $\quad$ are viewed as having the same number of variables, we have $\Pr[\,']\quad\Pr[\ ]$ by definition.

We also have the notion of induced formulas, obtained by asserting that certain literals must be true in a given CNF.

Definition 2.17 (Induced Formulas)**.** Let $'$ be a CNF. Let S be a subset of variables of $'$. Given an assignment $\quad : S \quad \{0,1\}$, we define $\quad '\quad$ to be the *induced formula* obtained by asserting values in $'$ according to $\quad$.

Formally, we construct $\quad '\quad$ by taking $'$, and for each variable $x \in S$:

- if $\quad(x) = 1$, remove each clause containing x and delete $\neg x$ from every clause of $'$, and
- if $\quad(x) = 0$, remove each clause containing $\neg x$ and delete x from every clause of $'$.

We then additionally add unit clauses with the literal x for each variable x with $\quad(x) = 1$, and unit clauses with the literal $\neg x$ for each variable $\neg x$ with $\quad(x) = 0$.

We view $\quad '\quad$ as a formula over the same set of variables of $'$. Given $\quad$ and $'$, the above description makes it clear that $\quad '\quad$ can be constructed in linear time.

Proposition 2.18. Let $'$ be a CNF formula. Let S be a subset of variables of $'$. For any assignment $\quad : S \quad \{0,1\}$, the set of satisfying assignments of $\quad '\quad$ is precisely the set of satisfying assignments of $'$ which agree with $\quad$ on S.

Proof. Fix an assignment $\quad : S \quad \{0,1\}$.

Take any satisfying assignment of $\quad '\quad$. Then this assignment must agree with $\quad$ on S, because of the unit clauses included in $\quad '\quad$ for each variable in S.

Each remaining clause of $\quad '\quad$ corresponds to a unique clause C in $'$, such that C does not contain any literal set to true by $\quad$. The clause in $\quad '\quad$ corresponding to a clause C in $'$ is C with all literals set to false by $\quad$ removed. So a satisfying assignment of $\quad '\quad$ must satisfy every clause of $'$ which does not have a literal set to true by $\quad$. Of course the clauses set to true by $\quad$ are also satisfied by the assignment, since it agrees with $\quad$ on S.

Thus a satisfying assignment of $\quad '\quad$ is a satisfing assignment for $'$.

Conversely, any satisfying assignment of $'$ which agrees with $\quad$ on S must satisfy $\quad '\quad$ as well. Indeed, all the unit clauses of $\quad '\quad$ added for variables of S are satisfied because the assignment agrees with $\quad$ on S. Each remaining clause B of $\quad '\quad$ is of the form $C \cap C^{\prime}$, where C is a clause in $\quad '\quad$, and $C^{\prime}$ is the set of literals in C set to false by $\quad$. An assignment which satisfies $'$ must satisfy some literal in C. If the assignment agrees with $\quad$ on S, then this literal must belong to $(C \cap C^{\prime}) = B$. Since this holds for all clauses B in $\quad '\quad$, we get that the assignment satisfies $\quad '\quad$ as claimed.

This shows the claimed equivalence, and proves the desired result.

Proposition 2.19. For any CNF formula $'$ and subset S of the variables of $'$, we have

$$\Pr['] = \sum_{: S \to \{0,1\}} \Pr[' \ ]:$$

Proof. Fix an assignment $: S \to \{0,1\}$. By Proposition 2.18, the satisfying assignments of $'$ are precisely the satisfying assignments of $'$ which agree with on S.

Since every assignment for $'$ assigns some values to the variables in S, the sets of satisfying assignments of each $'_j$ partition the set of satisfying assignments for $'$. Since each $'$ is viewed as a formula over the variables of $'$, $\Pr[' \ ]$ equals the number of satisfying assignments in $'$ divided by 2^n, where n is the number of variables in $'$. So

$$\Pr['] = \sum_{: S \to \{0,1\}} \Pr[' \ ]$$

as claimed.

Induced formulas arise from asserting that various literals are true in a CNF. It will also be helpful more generally to talk about formulas which arise from asserting that a whole clause must be true.

Proposition 2.20 (Asserting Clauses). There is an algorithm Assert, which takes as input a CNF formula $'$ and a clause C, and in linear time returns a CNF formula $= \text{Assert}('; C)$ equivalent to $' \wedge C$, such that the size of is at most the size of $' \wedge C$, contains the clause C, and contains no clause which is a proper superset of C.

Proof. We construct $= \text{Assert}('; C)$ as follows.

First scan through $'$, and identify the subformula

$$'_C = \{B \mid B \quad C; B \in '\}$$

of $'$ containing all clauses, and the subformula

$$'^{0} = \{B \mid B \not\in C; B \in '\}$$

of $'$ with the remaining clauses of $'$. Now construct the formula

$$= \{B \setminus C \mid B \in '_C\} \tag{1}$$

by removing C as a subset from every clause of $'_C$. We then return

$$= \quad \wedge '^{0} \wedge C \tag{2}$$

as the CNF formula $\text{Assert}('; C)$.

The algorithm runs in constant time, since we can construct just by scanning through the clauses of $'$ a constant number of times.

From eq. (2), contains the clause C. No clause in $'^{0}$ is a superset of C by construction, and no clause in is a superset of C by eq. (1), so no clause in is a proper superset of C by eq. (2). The formula has size less than or equal to the size of $' \wedge C$, because from eq. (2) we can construct by starting with $' \wedge C$ and deleting literals from some clauses. Finally, the formulas $\wedge C$ and $'_C \wedge C$ are equivalent by eq. (1), so the formulas and $' \wedge C$ are equivalent by eq. (2).

Thus the algorithm has the desired behavior.

Given a literal ℓ, we write

$$\mathsf{Assert}(\phi;\ell) = \mathsf{Assert}(\phi;f\ell g)$$

for convenience.

We only compute $\phi = \mathsf{Assert}(\phi;C)$ in cases where the clause C is over the underlying variable set of ϕ, and in these cases we view ϕ as having the same variable set as ϕ.

The following result lets us compute satisfaction probabilities using casework:

Proposition 2.21. For any $f;g\colon f0;1g^n \to f0;1g$ on the same set of variables, we have

$$\Pr[f] = \Pr[f \wedge g] + \Pr[f \wedge \neg g].$$

Proof. Every satisfying assignment of f either satisfies g or does not satisfy g. The number of satisfying assignments of f is $2^n \Pr[f]$. The number of these which satisfy g is $2^n \Pr[f \wedge g]$, and the number of these which do not satisfy g is $2^n \Pr[f \wedge \neg g]$. So

$$2^n \Pr[f] = 2^n \Pr[f \wedge g] + 2^n \Pr[f \wedge \neg g]$$

which implies the desired result.

We often apply Proposition 2.21 on the special case where g is a single literal ℓ.

2.4 Organization

In Chapter 3 we prove Theorem 2.15. Rather than prove the theorem in one shot, for a more accessible exposition we first prove easier versions of Theorem 2.15 when $k \geq 3$, pointing out the key ideas that go into developing our final algorithm. In Section 3.1 we show that $\mathsf{2SAT\text{-}Prob}_p$ can be solved in linear time for any constant $p \in (0;1)$, and in Section 3.2 show that $\mathsf{3SAT\text{-}Prob}_p$ can be solved in linear time for any constant $p \in (0;1)$. In Section 3.3 we prove Theorem 2.15, showing that $k\mathsf{SAT\text{-}Prob}_p$ can be solved in linear time for any constants $k \geq 1$ and $p \in (0;1)$. In Section 3.4 we provide some comments on this result, and use the proof of Theorem 2.15 to establish a structural theorem (which can be interpreted as a "regularity lemma") for k-CNF formulas.

In Chapter 4 we explore additional variants of the Majority-SAT problem, showing the surprisingly subtle nature of the complexity of threshold satisfaction problems. In Section 4.1 we design algorithms and prove hardness results for a variant of $k\mathsf{SAT\text{-}Prob}_p$, where instead of checking if $\Pr[\phi]$ is greater than or equal to p for a k-CNF ϕ, we are tasked with determining if $\Pr[\phi]$ is *strictly greater* than p. In Section 4.2 we show that Majority-SAT remains intractable even if we restrict the input formulas to contain at most *a single long clause* of width greater than k, where $k \geq 3$ is a constant. In Section 4.3 we consider a generalization of Majority-SAT involving *existential quantifiers*, which naturally arise when showing hardness results for probabilistic planning problems. In Section 4.4 we show hardness results for a variant of Majority-SAT related to *conditional probability*.

In Chapter 5, we conclude by stating relevant open problems in this area.

Chapter 3

Algorithms for Threshold Satisfiability

In this chapter, we present our algorithm for kSAT-Prob$_p$ and prove Theorem 2.15. Rather than launch directly into proving Theorem 2.15 however, we work our way up to this result by proving special cases of it for small k first.

3.1 Threshold 2SAT

In this section, we present a simple algorithm solving 2SAT-Prob$_p$ for constant $p \in (0;1)$ in linear time.

Theorem 3.1: Threshold Satisfaction for 2-CNFs

For any constant $p \in (0;1)$, there is a linear time algorithm solving 2SAT-Prob$_p$.

Our goal is to solve 2SAT-Prob$_p$ How should we do it? A natural starting point is to try and identify "easy cases" of the problem, where one can return YES or NO for simple reasons. One such easy NO instance comes from considering clauses on disjoint sets of variables.

Definition 3.2 (Disjoint Sets)**.** A set of clauses D is a *disjoint set* if no variable appears in two different clauses of D.

Proposition 3.3 (Large Disjoint Set $\Rightarrow$ Low Satisfaction Probability)**.** Let $'$ be a k-CNF with a disjoint set of size at least d. Then

$$\Pr[\ '\ ] \quad \left(1 \quad \frac{1}{2^k}\right)^d :$$

Proof. Let $D \quad '$ be a disjoint set of size at least d in $'$, viewed as a subformula of $'$.

An assignment fails to satisfy a clause precisely when it sets each literal of the clause to be false. This means that a clause C of width w is satisfied by a uniform random assignment with probability $\Pr[C] = 1 \quad (1=2)^w$, Since $'$ is a k-CNF, every clause in D therefore has satisfaction probability at most $1 \quad (1=2)^k$.

Since D is a disjoint set, each of its clauses being satisfied by a uniform random assignment are independent events. So $\Pr[D]$ is equal to the product of the satisfaction probabilities of

each of its clauses. Then by Proposition 2.16 we have

$$\Pr[\ '\ ] \quad \Pr[D] \quad \left(1 \quad \frac{1}{2^k}\right)^d$$

as claimed.

The result of Proposition 3.3 motivates the following idea for solving 2SAT-Prob $_p$.

> **Idea 2** Formulas with large disjoint sets have small satisfaction probability. So to solve the kSAT-Prob $_p$ problem, it should be useful to look for look for large disjoint sets.

To implement Idea 2, we need an algorithm for finding disjoint sets.

Proposition 3.4 (Finding Maximal Disjoint Sets). There is an algorithm Disj which takes as input a k-CNF formula $'$, and returns a maximal disjoint set $D = \mathrm{Disj}(')$ on the k-clauses of $'$ in linear time.

Proof. We construct the set D by greedily scanning through the clauses of $'$. Throughout, we will also maintain a set S of the variables occurring in clauses of D.

Initially, D ? consists of no clauses and S ? consists of no variables.

We go through the k-clauses of $'$ one at a time. When we encounter a clause C, we check if C contains any variables in S. If C does contain a variable from S, we move on to the next clause. Otherwise, if C contains no variables from S, then we add C to D and add the variables of C to S.

An easy induction argument shows that right after we encounter a clause, D is a disjoint set and S is precisely the set of variables contained in the clauses of D. After we have gone through all k-clauses in $'$, we return $\mathrm{Disj}(') = D$.

We claim that this algorithm has the desired behavior.

By the above discussion, D must be a disjoint set on the k-clauses of $'$.

B Claim 3.5. The set $D = \mathrm{Disj}(')$ is a *maximal* disjoint set on the k-clauses of $'$.

Proof. Suppose to the contrary D is not maximal on the k-clauses of $'$. Then there must exist a k-clause C of $'$ not in D, such that $D\ [\ fCg$ is a disjoint set.

The algorithm scans through all k-clauses of $'$. Since C was not included in the set D returned by the algorithm, it means that at the time we encountered C, this clause shared a variable with a clause already included in D. This contradicts the fact that $D\ [\ fCg$ is a disjoint set, so D is maximal on the k-clauses of $'$ as claimed. □

It remains to observe that the algorithm runs in linear time.

B Claim 3.6. Disj runs in linear time.

Proof. The algorithm goes through each k-clause C in $'$, and checks if C includes a variable from the currently used set of variables S. We can perform each such check in time linear in the size of C (e.g., we can do this by recording S using an array indexed by variables of $'$, where the entry for a variable x is set to 1 if and only if $x\ 2\ S$ and is zero otherwise), so the overall algorithm takes time linear in the size of $'$ as claimed. □

The desired results follows from Claims 3.5 and 3.6.

By combining Propositions 3.3 and 3.4, we should intuitively be able to find large disjoint sets in a k-CNF formulas if they exist, and return NO when solving kSAT-Prob$_p$ accordingly. What if the input formula has no large disjoint set? In this case, even though we cannot easily return an answer of NO, the lack of a large disjoint set is still some interesting information about the input formula which we can hope to leverage.

> **Idea 3** If a k-CNF $'$ avoids having a large disjoint set, its **clauses should overlap on a small set of variables**. We should exploit this structure to help simplify $'$.

The following result offers one way of formalizing the intuition from Idea 3.

Proposition 3.7 (Maximal Disjoint Set $)$ Hitting Set). Let $'$ be a k-CNF. Let D be a maximal disjoint set for the k-clauses of $'$. Let H be the set of variables appearing in some clause of D. Then every k-clause of $'$ contains a variable from H. In particular, for every assignment $: H \quad f0;1g$, the induced formula $'$ is a $(k \quad 1)$-CNF.

Proof. Suppose to the contrary that $'$ contains a k-clause C using no variables from H. Then $D \; [\; fCg$ is a disjoint set, which contradicts maximality of D. So the variables in H do indeed hit every k-clause in $'$.

Now take an arbitrary assignment $: H \quad f0;1g$. If C contains a literal that sets to true, then C is removed and does not appear in the induced formula $'$. Otherwise, every literal of C corresponding to a variable in H is set to false by . Every such literal is deleted from C to produce a clause included in $'$ The discussion from the previous paragraph shows that C uses a variable in H, so at least one literal is deleted in this case.

So in going from $'$ to $'$, each k-clause in $'$ is either removed or deletes at least one literal. Thus $'$ is a $(k \quad 1)$-CNF, because no k-clause from $'$ survives in $'$.

If our input k-CNF formula has no large disjoint set, then Proposition 3.7 says we can find a small set of variables H such that for every $: H \quad f0;1g$, the induced formula $'$ is a $(k \quad 1)$-CNF. So assignments to H intuitively help "simplify" $'$ to smaller width formulas. When $k = 2$, each $'$ is a 1-CNF. By Proposition 2.11, we can then compute satisfaction probabilities for $\Pr[' \;]$, and then compute $\Pr[']$ using Proposition 2.19.

Combining this discussion with the earlier arguments yields our approach for solving 2SAT-Prob$_p$, written out in Algorithm 1. At a high-level, Algorithm 1 works by combining Ideas 2 and 3 for 2-CNFs.

Lemma 3.8. Algorithm 1 solves 2SAT-Prob$_p$.

Proof. To prove the result, we step through Algorithm 1, and verify that it outputs the correct answer to the 2SAT-Prob$_p$ problem anywhere it returns YES or NO.

Let $d = d\log_{4=3}(1=p)e$ as in step 1 of Algorithm 1. By Proposition 3.4, the set D computed in step 2 of Algorithm 1 is a maximal disjoint set of 2-clauses in $'$.

If $jDj > d$, then by Proposition 3.3 we have

$$\Pr['] \quad \Pr[D] \quad (3=4)^{d+1} < p$$

Algorithm 1. Threshold Satisfaction Algorithm for 2-CNFs

Inputs: A 2-CNF $'$, and real $p \in (0;1)$.
Returns: YES if $\Pr[\,'\,] \geq p$, NO if $\Pr[\,'\,] < p$.

1. Set $d \leftarrow \lceil \log_{4/3}(1/p) \rceil$.
2. Compute $D \leftarrow \mathrm{Disj}(\,'\,)$.
3. If $|D| > d$, return NO.
4. Otherwise $|D| \leq d$. Let H be the set of variables appearing in D. Compute

$$\Pr[\,'\,] = \sum_{: H \to \{0,1\}} \Pr[\,'' \,]$$

and return YES if this sum is at least p, NO if this sum is less than p.

so the algorithm correctly returns NO in step 3 in this case.

Otherwise, by Proposition 2.19 the equation

$$\Pr[\,'\,] = \sum_{: H \to \{0,1\}} \Pr[\,'' \,]$$

from step 4 of Algorithm 1 holds, and the algorithm returns the correct answer in that step. So in all cases, Algorithm 1 returns the correct answer to the 2SAT-Prob $_p$ problem.

Proof of Theorem 3.1. By Lemma 3.8, Algorithm 1 correctly solves 2SAT-Prob $_p$. It remains to show that Algorithm 1 can be implemented to run in linear time.

Step 1 of Algorithm 1 takes constant time. By Proposition 3.4, constructing D in step 2 in Algorithm 1 takes linear time. Step 3 of Algorithm 1 involves checking $|D|$, which takes linear time since $|D| \leq |\,'\,|$.

We only reach step 4 of Algorithm 1 if $|D| \leq d$. In this case, since D consists of 2-clauses, we have $|H| \leq 2d$. For constant p we have $d \leq O(1)$, so constructing H takes constant time. Step 4 of Algorithm 1 ends by summing $\Pr[\,'' \,]$ over all $: H \to \{0,1\}$. Since $|H| \leq 2d$, there are at most $2^{2d} \leq O(1)$ choices of assignments . For each such , we can construct $''$ in linear time. By Proposition 3.7, each of these induced formulas is a 1-CNF, so we can compute each $\Pr[\,'' \,]$ in linear time by Proposition 2.11. So step 4 of Algorithm 1 takes linear time. Thus, Algorithm 1 takes linear time overall, which proves the desired result.

3.2 Threshold 3SAT

In this section, we present a simple algorithm solving 3SAT-Prob $_p$ for constant $p \in (0;1)$ in linear time. We design this algorithm by combining the simple ideas from Section 3.1 with more sophisticated observations about satisfaction probabilities of k-CNF formulas.

Theorem 3.9: Threshold Satisfaction for 3-CNFs

For any constant $p \in (0,1)$, there is a linear time algorithm solving 3SAT-Prob$_p$.

To solve 3SAT-Prob$_p$, a natural starting point is to try and apply Ideas 2 and 3, which were successful in solving 2SAT-Prob$_p$.

Given a 3-CNF ϕ, we can still return NO in the 3SAT-Prob$_p$ problem if ϕ contains a large disjoint set. If ϕ has no large disjoint set, then we can employ the idea from step 4 of Algorithm 1 and apply Proposition 3.7 to reduce computing $\Pr[\phi]$ to computing satisfaction probabilities for a small number of 2-CNFs ϕ' induced from ϕ.

Unfortunately, computing $\Pr[\phi]$ for 2-CNFs is #P-hard in general, so it is not immediately clear how to proceed from here. However, the idea of Algorithm 1 gives an example of a type of 2-CNF whose satisfaction probability we can find in polynomial time – namely, if the 2-CNF ϕ' has no large disjoint set, we can use step 4 of Algorithm 1 to compute $\Pr[\phi']$.

So if every ϕ' has no large disjoint set, we can compute $\Pr[\phi']$ for a small collection of assignments , and use this to compute $\Pr[\phi]$ exactly and solve 3SAT-Prob$_p$.

The only case the above strategy cannot handle is the situation where ϕ has no large disjoint set, yet some induced formula ϕ' does have a large disjoint set. A simple way this can occur is if the 3-CNF ϕ is of the form

$$\phi = f\ell \wedge [C \mid C \in D]g \tag{3}$$

where ℓ is a fixed literal and D is a disjoint set on 2-clauses not containing ℓ or $\neg\ell$ in any clauses. In this example, the maximum size of a disjoint set in ϕ is 1 (since all clauses share the common literal ℓ), but the assignment which sets ℓ to false induces a disjoint ϕ' of size $|D|$, which can be arbitrary large.

This structure, where a a large number of clauses overlap on a fixed set of literals but are disjoint otherwise, is a classic combinatorial object known as a sunflower.

Definition 3.10 (Sunflowers). Given a CNF ϕ, we say a set of clauses in ϕ is a *sunflower* if there exists a clause C such that

- every clause in is a superset of C, and
- the set $\{B \setminus C \mid B \in \}$ is a disjoint set.

We call $C = \text{core}( )$ the *core* of the sunflower . The number of literals $w = |C|$ in the core is called the *weight* of the sunflower. We define the *set of petals* of to be

$$\text{petals}( ) = \{B \setminus \text{core}( ) \mid B \in \}$$

the disjoint set formed by removing the core from each clause of . A sunflower of weight w is referred to as a *w-sunflower*.

So a disjoint set is a 0-sunflower, and the 3-CNF from eq. (3) is a 1-sunflower.

We saw in the previous discussion that eq. (3) is an example of a 3-CNF for which our strategy of computing $\Pr[\phi]$ by repeatedly applying Proposition 3.7 does not work. The next proposition shows that in some sense, **having a large sunflower is the only obstruction to finding $\Pr[\phi]$** by computing $\Pr[\phi']$ for various induced 1-CNF formulas ϕ'.

Proposition 3.11. For each integer $k \geq 1$, there is an algorithm Sun_k which takes as input a k-CNF formula φ and positive integer parameters $s_0, \ldots, s_{k-2}$, and returns either

1. a j-sunflower of size greater than s_j on w-clauses in φ for some $w \leq j+2$, or
2. a set H of at most $f_k(s_0, s_1, \ldots, s_{k-2})$ variables in φ such that for every $\tau : H \to \{0,1\}$, the induced formula $\varphi|_\tau$ is a 1-CNF, where $f_k(s_0, s_1, \ldots, s_{k-2})$ is a constant depending only on the s_j and k.

Moreover, the algorithm Sun_k runs in $f_k(s_0, \ldots, s_{k-2})|\varphi|$ time.

Proof. We prove the result by induction on k. Our argument is similar to the proof of the classic sunflower lemma of [ER60].

For $k = 1$, the algorithm Sun_1 simply returns the set of variables $H = \varnothing$, which satisfies item 2 from the statement because the input φ in this case is already a 1-CNF.

For the inductive step, let $k \geq 2$ be a fixed positive integer, and suppose we know there exists an algorithm Sun_{k-1} satisfying the desired properties for the $k-1$ case.

The algorithm Sun_k works as follows.

First, we compute a maximal disjoint set $D = \mathsf{Disj}(\varphi)$ on the k-clauses of φ using the linear-time algorithm from Proposition 3.4. If $|D| > s_0$, we return D, which satisfies item 1 from the statement since a disjoint set is a 0-sunflower.

Otherwise, $|D| \leq s_0$. In this case, let H be the set of variables which appear in a clause of D. Each clause in D contains k variables, so $|H| \leq ks_0$. Moreover, since D is maximal on the k-clauses of φ, every k-clause in φ contains some variable of H. So for each $\tau : H \to \{0,1\}$, the induced formula $\varphi|_\tau$ is a $(k-1)$-CNF.

We go through every assignment $\tau : H \to \{0,1\}$, and for each construct $\varphi|_\tau$ and compute

$$\mathsf{Sun}_{k-1}(\varphi|_\tau; s_0', s_1', \ldots, s_{k-3}') \tag{4}$$

where

$$s_j' = \left(\max_{r \leq j} s_r\right) \left(\sum_{i=0}^{k-2-j} (ks_0)^i\right) \tag{5}$$

for each $j \leq k-3$.

Note that the s_j' are constants depending only on $s_0, \ldots, s_{k-2}$.

We now perform casework based off what information the calls to Sun_{k-1} return.

Case 1: No Sunflowers

Suppose that for every assignment $\tau : H \to \{0,1\}$, the call to Sun_{k-1} in eq. (4) satisfies item 2 from the statement, returning a set H_τ of variables in $\varphi|_\tau$. We return

$$\bigcup_{\tau : H \to \{0,1\}} H_\tau.$$

We claim this output satisfies item 2 from the statement.

Indeed, since Sun_{k-1} satisfies item 2, we know that

$$|H_\tau| \leq f_{k-1}(s_0', \ldots, s_{k-3}')$$

for each assignment . As a consequence we have

$$jHj \quad 2^{ks_0} f_{k\ 1}(s_0^0; : : : ; s_{k\ 3}^0):$$

Now, set

$$f_k(s_0; : : : ; s_{k\ 2}) = 2^{ks_0} \quad f_{k\ 1}(s_0^0; : : : ; s_{k\ 3}^0): \tag{6}$$

Since each s_i^0 is a constant depending only on the s_j, by the inductive hypothesis the right hand side of the above expression is a constant depending only on the s_j and k.

So returning H satisfies item 2 from the statement as claimed.

Case 2: Some Sunflower

If case 1 does not occur, then for some assignment the output eq. (4) satisfies item 1 from the statement. This means that we have found a j-sunflower of size greater than s_j^0 in $'$, for some $j \quad k \quad 3$. Moreover, every clause in has width w for some $w \quad j + 2$.

Each w-clause in $'$ can be recovered uniquely by starting with a clause C of $'$ and removing at most $(k \quad w)$ literals from C. The only literals that are removed from clauses when going from $'$ to $'$ are those literals set to false by .

There are at most $jHj \quad ks_0$ such literals. So, there are at most

$$\sum_{i=0}^{k \quad w} (ks_0)^i \qquad \sum_{i=0}^{(k \quad 2) \quad j} (ks_0)^i$$

tuples of literals which could have been removed to obtain a clause in from a clause in $'$. Then by averaging, there exists a single set of literals S whose removal produces at least

$$\frac{s_j^0}{1 + (ks_0) + \quad + (ks_0)^{k \quad 2 \quad j}} = \max_{r \quad j} s_r \tag{7}$$

clauses in , where we used the definition of the s_j^0 from eq. (5).

Let ^{0} be the corresponding subformula of size at least $\max_{i \quad j} s_i$, consisting of the clauses in formed by deleting exactly the literals in S from clauses in $'$. We find ^{0} by going through the possible sets of at most $(k \quad w)$ literals set to false by , and for each seeing how many clauses in are obtained by removing exactly those literals from a clause in $'$.

Since is a j-sunflower, so is ^{0}.

Let $l = jSj$ be the number of literals in S. Note that

$$l \quad k \quad w \quad (k \quad 2) \quad j: \tag{8}$$

Consider the k-CNF formula with clause set

$$'^0 = fC \ [\ S \ j \ C \ 2 \quad ^0g:$$

We return $'^0$. We claim this output satisfies item 1 from the statement.

Since every clause in ^{0} was obtained by deleting exactly the literals in S from a clause in $'$, we know that $'^0 \quad '$ consists of clauses in $'$. Since ^{0} is a j-sunflower and no clause of ^{0} contains a literal from S, $'^0$ is a $(j + l)$-sunflower with $\mathrm{core}('^0) = S \ [\ \mathrm{core}(\ ^0)$. By eq. (8),

the weight of this sunflower is at most $j + l \quad k \quad 2$. Moreover, the formula $'^0$ consists entirely of clauses of width at least $(w + l) \quad (j + l) + 2$.

Finally, $'^0$ has the same size as ^{0}, which is at least s_{j+l} by eq. (7), so returning $'^0$ satisfies item 1 from the statement as claimed.

Thus in both case 1 and case 2, the output of Sun_k meets the conditions from the proposition statement. In either case, the runtime is dominated by the at most 2^{ks_0} calls to $\mathsf{Sun}_{k\ 1}$ in eq. (4). By the inductive hypothesis, each such call takes at most $f_{k\ 1}(s_0^0; : : : ; s_{k\ 3}^0)j\ 'j$ time asymptotically. Thus by eq. (5), Sun_k runs in

$$f_k(s_0; : : : ; s_{k\ 2})j\ 'j$$

time asymptotically as desired.

This completes the induction and proves the claim.

From Proposition 3.11, we see that given a 3-CNF $'$ we can, intuitively, in polynomial time either find a large 0-sunflower in $'$ (and therefore return NO to the $\mathsf{3SAT\text{-}Prob}_p$ problem), compute $\Pr['\ ]$ by computing $\Pr['\ \ ]$ for a small number of induced 1-CNFs $'$ (and solve the $\mathsf{3SAT\text{-}Prob}_p$ using the value of this satisfaction probability), or find a large 1-sunflower in $'$. What can we do in this last case?

Idea 3 suggests that if many clauses in $'$ overlap on a small set of literals, this structure should let us simplify $'$ somehow. A 1-sunflower is a very strong example of such structure: we have a set of clauses which all overlap on exactly one literal ℓ, and are disjoint otherwise. Intuitively, the literal ℓ is very important for satisfying $'$.

> **Idea 4** If a k-CNF has a large sunflower, then most of its satisfying assignments will also satisfy the the core of that sunflower. So restricting our attention to solutions of the original formula which satisfy the core of a large sunflower should still yield a good approximation to the true fraction of satisfying assignments.

Idea 4 is captured by the following result.

Proposition 3.12 (Falsifying the Core of a Large Sunflower). Let $'$ be a k-CNF containing a w-sunflower $\quad$ of size at least s. Let $C = \mathsf{core}(\ \ )$. Then

$$\Pr['\ \wedge\ :C] \quad \left(1 \quad \frac{1}{2^{k\ w}}\right)^s$$

Proof. Let $'^0$ be the formula obtained by taking $'$ and removing C from every clause in $'$ containing it as a subset. Then because any satisfying assignment to $'\ \wedge\ :C$ must falsify C, the formulas $'\ \wedge\ :C$ and $'^0\ \wedge\ :C$ are equivalent.

Since $\quad '$ has core C, by construction $\mathsf{petals}(\ \ ) \quad '^0$. Hence $'^0$ contains a disjoint set of size at least s. So by Propositions 3.3 and 2.16 we have

$$\Pr['^0] \quad \left(1 \quad \frac{1}{2^{k\ w}}\right)^s$$

where we used the fact that $\mathsf{petals}(\ \ )$ is a $(k \quad w)$-CNF.

Every solution to $'^0 \wedge C$ is a satisfying assignment of $'^0$. Thus

$$\Pr[' \wedge :C] = \Pr['^0 \wedge :C] \quad \Pr['^0] \quad \left(1 \quad \frac{1}{2^{k \ w}}\right)^s$$

which proves the claim.

We can apply Proposition 3.12 to 3SAT-Prob $_p$ as follows. If our 3-CNF $'$ has a large 1-sunflower with core $f`g$, then since every solution to $'$ sets $`$ to true or false we have

$$\Pr[\,'\,] = \Pr[' \wedge `] + \Pr[' \wedge :`]. \tag{9}$$

The formula $' \wedge :`$ is equivalent to the induced formula $'$ , where is the assignment which sets $`$ to false. Since $`$ is the core of a large 1-sunflower in $'$, the formula $'$ has a large disjoint set, and thus $\Pr[' \ ]$ is small. So by eq. (9), $\Pr[\,']$ should be very close to $\Pr[' \wedge `]$. The formula $' \wedge `$ is equivalent to formula induced from $'$ by assigning $`$ to be true, which erases the large 1-sunflower from $'$, and so intuitively "simplifies" $'$. Thanks to this simplification, working with $' \wedge `$ should be easier than working with $'$ directly.

If $\Pr[' \wedge `]$ p, then by eq. (9) we know that $\Pr[\,']$ p as well, and can return YES in the 3SAT-Prob $_p$ problem.

What can we deduce if $\Pr[' \wedge `] < p$ instead? In this case, from eq. (9) alone it is not necessarily clear how $\Pr[\,']$ compares to $p$. Even if $\Pr[' \wedge :`] <$ is very small, if somehow we have $\Pr[' \wedge `]$ 2 $(p \quad ; p)$, then it could still be the case that $\Pr[\,']$ p.

We will argue that for a small enough constant > 0, it is actually *impossible* for the above situation to occur. This is suggested by Idea 1: the possible satisfaction probabilities of k-CNFs seem highly constrained for constant k, so intuitively it should not be possible for $\Pr[' \wedge `]$ to be both less than p and very close to p.

To make this intuition more concrete, suppose for simplicity that $' \wedge `$ contains no large sunflower, so that by Proposition 3.11 the solution space of $' \wedge `$ can be decomposed into a small union of solution spaces of 1-CNFs. By Proposition 2.11, the satisfaction probability for a 1-CNF is zero or a power of two. So in this case, $\Pr[' \wedge `]$ is a sum of a small number of powers of two. The following lemma shows that such a number cannot be arbitrarily close to p and yet less than p, for any constant p $(0; 1)$.

Lemma 3.13 (Binary Gaps). For any p $(0; 1)$ and positive integer m, there exists a real number $= {}_m(p) > 0$ such that no real in $(p \quad ; p)$ is the sum of at most m powers of two.

Proof. We use the following well-known fact: any real number p $(0; 1)$ has a unique binary representation of the form

$$p = \sum_{i=1}^{1} 2^{a_i} \tag{10}$$

where $(a_i)_{i \ 1}$ is a sequence of strictly decreasing integers. Define

$$= {}_m(p) = \sum_{i=m+1}^{1} 2^{a_i} \tag{11}$$

to be number obtained by taking all but the first m terms in the summation representing p. We claim that no number which can be written as the sum of at most m powers of two is in the interval $(p \quad ; p)$.

To that end, take an arbitrary real q which can be written as the sum of at most m powers of two. Let $l \quad m$ be the smallest positive integer such that q can be written as the sum of l powers of two. Take such a representation

$$q = \sum_{i=1}^{l} 2^{b_i}$$

for some integers $b_1 \quad b_2 \qquad b_l$. We claim that $(b_i)_{i \ 1}$ is a strictly decreasing sequence. Indeed, if not, then there exists an index j for which $b_j = b_{j+1}$. But then since

$$2^{b_j} + 2^{b_{j+1}} = 2 \ \ 2^{b_j} = 2^{b_j+1}$$

we get that q can be written as the sum of $l \quad 1$ powers of two, which contradicts our choice of l. So we must have $b_1 > b_2 > \qquad > b_l$ as claimed.

We now compare the binary digits of q and $\quad$ compare.

Suppose first that $b_i = a_i$ for all $i \ 2 \ [l]$. Then since $l \quad m$, we have

$$q = \sum_{i=1}^{l} 2^{b_i} = \sum_{i=1}^{l} 2^{a_i} \quad \sum_{i=1}^{m} 2^{a_i} = p \quad :$$

Otherwise, $b_i \neq a_i$ for some $i \ 2 \ [l]$. Let $j \ 2 \ [l]$ be the smallest index with $b_j \neq a_j$.

If $b_j < a_j$, then

$$q = \sum_{i=1}^{l} 2^{b_i} \quad \left(\sum_{i=1}^{j \ 1} 2^{a_i} \right) + 2^{a_j \ 1} + \sum_{r=1}^{1} 2^{a_j \ 1 \ r} = \sum_{i=1}^{j} 2^{a_i}$$

where we have used the infinite geometric series formula and the fact that $(a_i)_{i \ 1}$ is a strictly decreasing sequence. Since $j \quad l \quad m$, we see that in this case

$$q \quad \sum_{i=1}^{j} 2^{a_i} \quad \sum_{i=1}^{m} 2^{a_i} = p$$

as before.

The only remaining possibility is that $b_j > a_j$. In this case we have

$$q \quad \sum_{i=1}^{j} 2^{b_i} \quad \left(\sum_{i=1}^{j \ 1} 2^{a_i} \right) + 2^{a_j+1} = \sum_{i=1}^{j} 2^{a_i} + \sum_{r=1}^{1} 2^{a_j \ r} \quad p$$

where we have once again used the infinite geometric series formula and that the a_i are strictly decreasing, as well as eq. (10).

So in every case, we have $q \notin (p \quad ; p)$ as claimed.

Incorporating Lemma 3.13 into the previous discussion on how to solve $\mathsf{3SAT\text{-}Prob}_{\ p}$ on input $'$, we have the following approach to tackle the problem:

1. If φ has a large disjoint set we return NO.

2. If $\Pr[\varphi]$ can be computed as a small sum of satisfaction probabilities of 1-CNFs, we can also solve the problem.

3. If neither of these cases occur, then φ contains a large 1-sunflower by Proposition 3.11. If $\text{core}(\quad) = f \ell g$, we have $\Pr[\varphi] = \Pr[\varphi \wedge \ell] + \Pr[\varphi \wedge \neg \ell]$. Since ℓ is the core of a large sunflower, $\Pr[\varphi \wedge \neg \ell] <$ is very small.

4. If $\Pr[\varphi \wedge \ell]$ can be computed as a small sum of satisfaction probabilities of 1-CNFs, then by Lemma 3.13, $\Pr[\varphi] \quad p$ if and only if $\Pr[\varphi \wedge \ell] \quad p$, so we can solve the problem.

The application of Lemma 3.13 in step 4 above is a concrete way of realizing Idea 4: the literal ℓ is so influential in φ, that for the purpose of checking if $\Pr[\varphi] \quad p$ it suffices to work with the "inferred formula" $\varphi \wedge \ell$ and check if $\Pr[\varphi \wedge \ell]$ instead.

It remains to handle the case where in item 4 above, $\Pr[\varphi \wedge \ell]$ is not equal to a small sum of satisfaction probabilities of 1-CNFs. In this case, by Proposition 3.11 the formula $\varphi \wedge \ell$ has a large sunflower. Then we can repeat our strategy above: use this large sunflower to either report that $\Pr[\varphi \wedge \ell]$ is very small (so we can return NO) or simplify $\varphi \wedge \ell$ by asserting the core of the sunflower is true.

We keep repeating this sort of simplification as needed, inferring simpler formulas each time. Eventually, either we simplify to a formula whose satisfaction probability we can compute exactly to solve the problem, or we have find so many different 1-sunflowers in the original formula φ that we can argue that $\Pr[\varphi] < p$.

> **Idea 5** If a CNF formula contains many large, non-overlapping sunflowers, its satisfaction probability should be very small.

This strategy is implemented in Algorithm 2 to solve 3SAT-Prob$_p$.

In Algorithm 2, d is the threshold for a disjoint set to be large, t is the number of large 1-sunflowers we need to see before we can deduce that φ has a small satisfaction probability (following Idea 5), and each $s(i)$ input is the size threshold for the i^{th} 1-sunflower we find to be considered large.

Lemma 3.14. For any $p \; 2 \; (0; 1)$, there exist positive integers $s(i)$ for each $i \; 2 \; [d\log(1=p)e]$ whose values depend only on i and p, such that Algorithm 2 solves 3SAT-Prob$_p$ when given the $s(i)$ as input.

Proof. We first describe the values of the $s(i)$, and then prove that this choice of parameters enables Algorithm 2 to correctly solve 3SAT-Prob$_p$.

As in Algorithm 2, we write $t = d\log(2=p)e$ and $d = d\log_{8=7}(2=p)e$

In addition to the $s(i)$, we define integers

$$m_i = 2^{f_3(d;s(i))} \tag{12}$$

for each $i \; 2 \; [t]$, where f_3 is the function f_k from the statement of Proposition 3.11 for $k = 3$. Since d depends only on p, each m_i is determined by the value of p and $s(i)$.

Algorithm 2. Threshold Satisfaction Algorithm for 3-CNFs

Inputs: A 3-CNF $'$, real $p \in (0;1)$, and integers $s(i)$ for each $i \in [d\log(1=p)e]$
Returns: YES if $\Pr[\,'\,] \geq p$, NO if $\Pr[\,'\,] < p$

1. Initialize $\quad '$, $t \quad d\log(2=p)e$, $d \quad d\log_{8=7}(2=p)e$.
2. For each $i \in [t]$:
3. Compute $S \quad \mathrm{Sun}_3(\ ;d;s(i))$
4. If $D \quad S$ is a disjoint set of size greater than d, return NO.
5. If instead $H \quad S$ is a collection of variables, compute

$$\Pr[\ ] = \sum_{: H \to \{0,1\}} \Pr[\ ]$$

   and return YES if this sum is at least p, NO if this sum is less than p.
6. Otherwise $\quad S$ is a 1-sunflower of size greater than $s(i)$.
   Set $C \quad \mathrm{core}(\ )$. Update $\quad \mathrm{Assert}(\ ;C)$.
7. Return NO.

We set

$$s(t) = d\log_{4=3}(2t=p)e: \tag{13}$$

Then for each index i with $2 \quad i \quad t$, we set

$$s(i \quad 1) = \left\lceil \log_{4=3}\left(\frac{i \quad 1}{m_i(p)}\right)\right\rceil \tag{14}$$

where $_m(p)$, for any integer m, is the constant from the statement of Lemma 3.13. Since each m_i is defined in terms of p and $s(i)$, the above equation defines $s(i \quad 1)$ in terms of $s(i)$ for each $2 \quad i \quad t \quad 1$. So eqs. (12) to (14) define the $s(i)$ values for all $i \in [t]$.

We now show that for this setting of parameters, Algorithm 2 has the desired behavior.

Suppose that Algorithm 2 reaches some iteration i of the loop. Then for each $j < i$, the call to Sun_3 in iteration j of the loop must of have returned a 1-sunflower of size greater than $s(j)$ in , since otherwise the algorithm would have halted. Let $`_j$ be the literal making up the core of the 1-sunflower found in iteration j.

Since $= '$ at the beginning of the algorithm, and in each iteration j we update to be equivalent to $\wedge `_j$, an easy induction argument shows that at the end of iteration j, the formula is equivalent to

$$' \wedge \left(\bigwedge_{r=1}^{j} `_r\right):$$

So immediately after Algorithm 2 has completed i iterations of the loop, is equivalent to

$$' \wedge \left(\bigwedge_{j=1}^{i} \grave{}_j \right) : \tag{15}$$

Then at this time, the satisfying assignments of are precisely the solutions to $'$ which set $\grave{}_j$ to be true for all $j < i$. This means that the assignments which satisfy $'$ but do not satisfy assign $\grave{}_j$ to be false for some index j. By considering the smallest index j which satisfies this property for each assignment, we deduce that

$$\Pr[\,'\,] \quad \Pr[\ \ ] = \sum_{j=1}^{i} \Pr\left[' \wedge \left(\bigwedge_{r=1}^{j} \grave{}_r \right) \wedge : \grave{}_j \right] : \tag{16}$$

For each $j < i$, we know that Algorithm 2 found a sunflower of size greater than $s(j)$ in a formula equivalent to

$$' \wedge \left(\bigwedge_{r=1}^{j} \grave{}_r \right)$$

with core $f \grave{}_j g$. So by Proposition 3.12,

$$\Pr\left[' \wedge \left(\bigwedge_{r=1}^{j} \grave{}_r \right) \wedge : \grave{}_j \right] < (3=4)^{s(j)}$$

Substituting the above inequality for each $j < i$ into eq. (16) we get that

$$\Pr[\,'\,] \quad \Pr[\ \ ] < (i \quad 1)(3=4)^{s(i \ \ 1)} \tag{17}$$

where we used the fact that the $s(j)$ are decreasing.

Having proved $\Pr[\,']$ and $\Pr[\ \ ]$ are close, we are ready to analyze what happens when Algorithm 2 halts. We consider cases based off whether Algorithm 2 halts within its loop (in steps 4 or 5), or outside of its loop (in step 7).

Case 1: Halting Within the Loop

Suppose Algorithm 2 halts on iteration i of the loop.

If we halt in step 4 of Algorithm 1, has a disjoint set of size greater than d. Then by Proposition 3.3 we have

$$\Pr[\ \ ] < (7=8)^{d} \quad p=2$$

since $d > \log_{8=7}(2=p)$.

By eq. (17) and the fact that the $s(i)$ are decreasing we have

$$\Pr[\,'\,] \quad \Pr[\ \ ] < t \ \ (3=4)^{s(t)} \quad p=2$$

by our choice of $s(t)$ in eq. (13).

Adding the two last equations together, we deduce that

$$\Pr[\,'\,] = \Pr[\ \ ] + (\Pr[\,'\,] \quad \Pr[\ \ ]) < p=2 + p=2 = p$$

so Algorithm 2 correctly returns NO in this case.

The other possibility is that we halt in step 5 of Algorithm 1. In this case, by Proposition 3.11, we find a set H of at most $f_3(d; s(i))$ variables in with the property that for every assignment : H $f0; 1g$, the induced formula is a 1-CNF.

By Proposition 2.19 we have

$$\Pr[\ ] = \sum_{: H \ \square \ f0;1g} \Pr[\ ]:$$

By Proposition 2.11, each summand in the right hand side above is zero or a power of two. So the above equation shows that $\Pr[\ ]$ is a sum of at most

$$2^{jHj} \quad 2^{f_3(d;s(i))} = m_i$$

powers of two by our choice of m_i in eq. (12).

If $\Pr[\ ]$ p, then since is equivalent to the formula from eq. (15), we have

$$\Pr[\ '] \quad \Pr[\ ] \quad p$$

so Algorithm 2 returns YES correctly in this case.

If $\Pr[\ ] < p$, by Lemma 3.13 we have

$$\Pr[\ ] \quad p$$

for = $_{m_i}(p)$.

From eq. (17) and our choice of $s(i)$ in eq. (14), we can bound

$$\Pr[\ '] \quad \Pr[\ ] < (i \quad 1)(3=4)^{s(i \ \ 1)} \quad :$$

Adding the previous two inequalities yields

$$\Pr[\ '] = \Pr[\ ] + (\Pr[\ '] \quad \Pr[\ ]) < (p \quad) + \quad = p$$

so Algorithm 2 returns NO correctly in this case.

Thus Algorithm 2 returns the correct answer whenever it halts within the loop.

Case 2: Halting Outside the Loop

In this case, Algorithm 2 halts in step 7.

This means the algorithm completed t iterations of the loop. So by eq. (17) we have

$$\Pr[\ '] \quad \Pr[\ ] < t(3=4)^{s(t)} \quad p=2$$

by our choice of $s(t)$ in eq. (13).

By eq. (15), we know that is equivalent to

$$' \ \wedge \left(\bigwedge_{j=1}^{t} \ `_j \right) :$$

Each ℓ_j was identified as the core of a large sunflower in a formula containing only one instance of ℓ_r (as a unit clause) for all $r < j$. Thus $D = \{\ell_1, \ldots, \ell_t\}$ is a disjoint set in the above formula. Then by Proposition 2.16 we have

$$\Pr[\ ] \quad \Pr[D] \quad (1/2)^t \quad p/2$$

by our choice of $t > \log(2/p)$.

Combining the previous two inequalities, we get that

$$\Pr[\ '] = \Pr[\ ] + (\Pr[\ '] \quad \Pr[\ ]) < p/2 + p/2 = p$$

so Algorithm 2 returns NO correctly in this case.

So in both case 1 and case 2, Algorithm 2 solves 3SAT-Prob$_p$ correctly as claimed.

Proof of Theorem 3.9. By Lemma 3.14, for any constant $p \in (0, 1)$, there exist constant positive integers $s(i)$ for each $i \in [d\log(2/p)e]$ such that Algorithm 2 solves 3SAT-Prob$_p$ when given the $s(i)$ as input. So to prove the theorem, it suffices to show that Algorithm 2 runs in linear time when given these input parameters.

Step 1 of Algorithm 2 takes linear time since we just read the input.

There are at most $t = d\log(2/p)e$ iterations of the loop in Algorithm 2. Since p is constant, t is bounded above by a constant. So steps 3 through 6 of the algorithm each occur at most a constant number of times.

By Proposition 3.11, the call to Sun_3 in step 3 of Algorithm 2 takes linear time, since the $s(i)$ are constants and $d = d\log_{8/7}(2/p)e$ is constant because p is constant.

Step 4 of Algorithm 2 takes linear time since we just need to read a disjoint set in .

Step 5 of Algorithm 2 takes at most $2^{jHj}j\ 'j$ time asymptotically. By Proposition 3.11 the number of variables in H is bounded above by a constant since d and $s(i)$ are constants, so this step takes linear time.

Step 6 of Algorithm 2 takes linear time by Proposition 2.20.

Step 7 of Algorithm 2 takes constant time since we just return NO.

So overall Algorithm 2 takes linear time, as claimed.

3.3 Threshold kSAT

In this section, we show how to solve kSAT-Prob$_p$ for fixed integers $k \quad 1$ and constant $p \in (0, 1)$ in linear time, thereby proving Theorem 2.15. We do this by generalizing the ideas from Sections 3.1 and 3.2 to k-CNF for $k \quad 4$.

The general strategy of our kSAT-Prob$_p$ algorithm is the same as that of Algorithm 2. We keep applying Proposition 3.11 to our formula. If the Sun_k subroutine from Proposition 3.11 finds a large disjoint set or lets us compute the satisfaction probability of the formula exactly, we can solve the problem. Otherwise, Sun_k finds a large sunflower in the formula. In this case, we update the formula by asserting that the core of its large sunflower is true. Intuitively, as suggested in Idea 4, such an overwhelming amount of the satisfying assignments to $'$ set this core to be true that asserting the core is true is a "safe inference" for the purpose of solving kSAT-Prob$_p$. We keep doing this until the above strategy returns an answer, or we have found "too many" large sunflowers, at which point we can return NO.

The proof of correctness for our kSAT-Prob$_p$ algorithm when $k \geq 4$ becomes more involved than the arguments in Sections 3.1 and 3.2 for the $k = 3$ case because the sunflowers we find can how have cores with more than one literal. This means that quantifying precisely how "large" each sunflower should be becomes trickier to analyze (in particular, the dependence between parameter values becomes more difficult to keep track of), and that arguing that we can return NO if we find many sunflowers becomes harder because the cores of different sunflowers can overlap.

When we find "many" large sunflowers, we saw in Section 3.2 that when solving 3SAT-Prob$_p$ we can just return NO, intuitively because of Idea 5. For this intuition to work we need to find sunflowers which do not overlap at many literals, so that we can argue the formula has small satisfaction probability.

To circumvent this issue, we modify the algorithm as follows: any time Sun$_k$ finds a large w-sunflower with core C for some $w \geq 1$, we try out each nonempty $B \subseteq C$ and check if B is the core of a v-sunflower larger than in the formula, where $v = |B| < w$. If such a set B exists, then we assert that B is true instead of asserting that C is true.

Why is this change helpful? Well, this check ensures that whenever we assert that the core of a large sunflower is true, we know that none of its literals belong to cores of large sunflowers with smaller weight. Intuitively, this means that the literals in the core we asserted should not be able to overlap too much with the cores of large sunflowers we find later (since if there were many large overlaps, we should have a sunflower with smaller core in the above check and asserted that instead).

To perform this check it is not enough to rely on the Sun$_k$ routine from Proposition 3.11 as written, since we do not have control over the weight of the sunflower returned in Sun$_k$. For our kSAT-Prob$_p$ algorithm here, we will implement the check using an existing parameterized algorithm for a certain maximum disjoint set problem.

In the (k, s)-Set Packing problem, we are given a family F of sets, each of size exactly k, and are tasked with returning a collection of s pairwise-disjoint sets in F, or reporting that no such collection exists.

The following result is due to [JZC04], and we cite it here without proof:

Proposition 3.15 (Parameterized Set Packing Algorithm). The (k, s)-Set Packing problem can be solved in linear time for constant positive integers k and s.

Using the algorithm from Proposition 3.15, we can run the check discussed previously.

Corollary 3.16. For each integer $k \geq 1$, there is an algorithm MaxSun$_k$ which takes as input a k-CNF , an integer $s \geq 1$, and a clause B, and either returns a sunflower of size greater than s, with core( ) $= B$, in , or reports that no such sunflower exists (i.e., returns NO). If k and s are constants, the algorithm runs in linear time.

Proof. The MaxSun$_k$ algorithm works by reducing to the (k, s)-Set Packing problem. We will first describe the algorithm, and then explain why it works.

Given , we first construct the subformula

$$' = \{C \mid C \in 2 \quad ; C \quad B\}$$

of all clauses in containing B as a subset. Then we construct the formula

$$' = fC \; n \; B \; j \; C \; 2 \; 'g$$

obtained by removing the literals of C from each clause in $'$. We can construct $'$ in linear time simply by scanning through the clauses of .

We then construct a family F of sets by taking the clauses of $'$, and for each variable x in $'$ replacing all instances of $:x$ with x in the clauses. So F is the set family obtained by identifying literals corresponding to the same variable as the same elements in the clauses.

Since is a k-CNF, every set in F contains at most k elements.

We construct the family of sets F by taking F and for each set $S \; 2 \; F$ containing fewer than k elements, adding $k \quad jSj$ new elements to that set, not included in any other set. We can construct F in linear time by scanning through the clauses and variables of $'$. Note that each set in F corresponds uniquely to a clause in $'$, and we record this correspondence in linear time as well.

By construction, F is a set family where every set has size exactly k.

We now run the algorithm from Proposition 3.15 on F with parameters k and $s+1$. This takes linear time since k and s are constants.

If the set packing algorithm says that F does not contain $s+1$ pairwise disjoint sets, we report that does not contain a sunflower of size greater than s with core B.

If instead the set packing algorithm returns a collection D of $s+1$ pairwise disjoint sets in F, we find the set of clauses D in $'$ corresponding to D (using the correspondence we kept track of) and then return

$$= fC \; [\; B \; j \; C \; 2 \; Dg$$

as our sunflower with core B, of size greater than s.

The discussion above shows that the proposed algorithm runs in linear time. It remains to explain why the algorithm is correct.

First, a collection of sets D in F are pairwise disjoint if and only if the corresponding collection of sets in F (before new elements were added to make all sets have size exactly k) are pairwise disjoint. This is because the newly added elements in going from F to F were different for every set, and adding new elements to sets which intersect cannot turn the sets disjoint. It then follows from the definition of a disjoint set of clauses in Definition 3.2 that a collection of sets D in F are pairwise disjoint if and only if the corresponding set of clauses D in $'$ is variable disjoint. Moreover, D and D have the same size.

If contains a sunflower with core B, of size greater than s, then petals( ) form a disjoint set of size greater than s in $'$. Similarly, if $'$ contains a disjoint set D of size greater than s in $'$, we can pull back D to a get a sunflower

$$= fC \; [\; B \; j \; C \; 2 \; Dg$$

of size greater than s, with core B, in .

The discussions from the previous two paragraphs show that the proposed algorithm has the desired behavior, which completes the proof.

To help argue that we can return NO once our algorithm has found enough large sunflowers, we use the following variant of the sunflower lemma from [ER60].

Proposition 3.17 (Asymmetric Sunflower Lemma). Let w be a positive integer. Then for any positive integers $s_0, \ldots, s_{w-1}$, every w-CNF formula with more than

$$w! \cdot 2^w \prod_{v=0}^{w-1} s_v$$

clauses must contain a v-sunflower of size greater than s_j for some $j \in [w]$.

Proof. We prove the result by induction on w.

For the base case of $w = 1$, we want to show that a 1-CNF φ with more than $2s_0$ clauses contains a disjoint set of size greater than s_0. A variable x can appear in φ only as a unit clause $\{x\}$ or a unit clause $\{\bar{x}\}$. So each variables appears in at most twice in φ. Since φ has more than $2s_0$ clauses, it must contain more than $2s_0/2 = s_0$ variables. The clauses containing these distinct variables form a disjoint set of size greater than s_0 as desired.

For the inductive step, fix $w \geq 2$, and suppose we know that the proposition statement holds for $(w-1)$-CNFs. We will show that any w-CNF φ with more than

$$w! \cdot 2^w \prod_{v=0}^{w-1} s_v$$

clauses contains a v-sunflower of size greater than s_v for some $v < w$.

Let D be a maximum size disjoint set in φ.

If $|D| > s_0$ then the claim holds.

Otherwise, $|D| \leq s_0$. Let H be the set of variables appearing in D. Since φ is a w-CNF, we have $|H| \leq ws_0$. Since D must also be a *maximal* disjoint set in φ, every clause in φ use a variable from H. There are $2|H| \leq 2ws_0$ literals corresponding to the variables in H. By averaging, there must be a literal ℓ corresponding to a variable from H which appears in more than

$$\frac{1}{2ws_0} \cdot w! \cdot 2^w \prod_{v=0}^{w-1} s_v = (w-1)! \cdot 2^{w-1} \prod_{v=0}^{w-2} s_{v+1} \tag{18}$$

clauses of φ. Let ψ be the subformula of φ with all clauses containing ℓ. Let

$$\varphi' = \{C \setminus \{\ell\} \mid C \in \psi\}$$

be the $(w-1)$-CNF obtained by removing ℓ from each clause of ψ.

Since the number of clauses in φ' is greater than the right hand side of eq. (18), by the inductive hypothesis there exists $v < w-1$ such that φ' contains a v-sunflower of size greater than s_{v+1}. Let S' be such a sunflower. Then since no clause in S' contains ℓ, the formula

$$S = \{C \cup \{\ell\} \mid C \in S'\}$$

is a $(v+1)$-sunflower in φ of size greater than s_{v+1}.

Note that $v + 1 < w$ since $v < w - 1$, so the proposition statement holds for w.

This completes the induction and proves the desired result.

The last concept we introduce before presenting the kSAT-Prob$_p$ algorithm is that of a parameter family. A parameter family consists of sequences of t_w and s_w of parameters for each $w \in [k-2]$, indexed by tuples $\vec{i} = \langle i_1, \ldots, i_{k-2} \rangle$. Intuitively, in our algorithm i_w will record the number of large w-sunflowers found so far for each $w \in [k-2]$ Given these counts, $t_w(\vec{i})$ will record the maximum number of additional large w-sunflowers we can find before our algorithm halts and returns NO, and $s_w(\vec{i})$ represents the current threshold for a w-sunflower to be large.

Definition 3.18 (Parameter Family)**.** Given an integer $k \geq 3$ and real $p \in (0,1)$, we define a (k,p)-parameter family $P = (t_1, s_1, \ldots, t_{k-2}, s_{k-2})$ to be a collection of sequences of positive integers with the following properties:

1. Sequence $t_1 = \langle t_1(?) \rangle$ consists of a single positive integer $t_1(?)$.
2. For each $w \geq 2$, t_w is a sequence of integers indexed by $(w-1)$-tuples $\langle i_1, \ldots, i_{w-1} \rangle$ of integers satisfying $0 \leq i_v < t_v(i_1, \ldots, i_{v-1})$ for each $v \in [w-1]$.
3. For each $w \in [k-2]$, s_w is a sequence of integers indexed by w-tuples $\langle i_1, \ldots, i_w \rangle$ of integers satisfying $0 \leq i_v < t_v(i_1, \ldots, i_{v-1})$ for each $v \in [w]$.

We say a $(k-2)$-tuple $\vec{i} = \langle i_1, \ldots, i_{k-2} \rangle$ which satisfies

$$0 \leq i_w < t_w(i_1, \ldots, i_{w-1}) \tag{19}$$

for all $w \in [k-2]$ is a *valid index* for the parameter family P. We let $I(P)$ denote the set of valid indices for P.

For each $w \in [k-2]$ and valid index $\vec{i} = \langle i_1, \ldots, i_{k-2} \rangle$ for P, we define

$$t_w(\vec{i}) = t_w(i_1, \ldots, i_{w-1}) \quad \text{and} \quad s_w(\vec{i}) = s_w(i_1, \ldots, i_w).$$

If $\vec{i} = \langle i_1, \ldots, i_{w-1} \rangle$ is outside the index set for t_w, we set $t_w(\vec{i}) = 0$ by convention.

To argue our algorithm runs in linear time, we will need to be able to bound the number of valid indexes for a given (k,p)-parameter family. To do this, it is helpful to upper bound the number of possible *prefixes* of a valid indices.

Definition 3.19 (Valid Index Prefixes)**.** Let $P = (t_1, s_1, \ldots, t_{k-2}, s_{k-2})$ be a (k,p)-parameter family for some positive integer $k \geq 3$ and $p \in (0,1)$. Then for any $w \in [k-2]$, we say a tuple of integers $\vec{j} = \langle j_1, \ldots, j_w \rangle$ is a *valid w-prefix* for P if

$$0 \leq j_v < t_v(j_1, \ldots, j_{v-1}) \tag{20}$$

for all $v \in [w]$. In particular, $\vec{j}$ is a valid $(k-2)$-prefix precisely if it is a valid index of P. We let $I_w(P)$ denote the set of valid w-prefixes of P.

Lemma 3.20 (Bounding Valid Prefix Count)**.** Let $P = (t_1, s_1, \ldots, t_{k-2}, s_{k-2})$ be a (k,p)-parameter family for some positive integer $k \geq 3$ and $p \in (0,1)$. Then for any $w \in [k-2]$, the number of valid w-prefixes of P is at most

$$T_w = \prod_{v=1}^{w} \left(\max_{\vec{t} \in I_{v-1}(P)} t_v(\vec{t}) \right).$$

Proof. We prove the result by induction on w.

For the base case of $w = 1$, a valid w-prefix $\vec{j} = \langle j_1 \rangle$ consists of a single nonnegative integer with $j_1 < t_1(?)$. So there are at most $t_1(?)$ valid 1-prefixes. Since $I_0(P) = \{?\}$, this proves the claim for $w = 1$.

For the inductive step, fix $w \in [k \quad 2]$ with $w \quad 2$, and suppose the claim holds for the case of $(w \quad 1)$. We will show that the claim holds in the case of w as well.

Let $\vec{j} = \langle j_1; \ldots; j_w \rangle$ be a valid w-prefix. Then $\vec{j}$ satisfies the inequalities from eq. (20) for each $v \in [w]$. This implies that $\vec{t} = \langle j_1; \ldots; j_{w \ 1} \rangle$ is a valid $(w \quad 1)$-prefix. By the inductive hypothesis, there are at most $T_{w \ 1}$ such $\vec{t}$. For each choice of $\vec{t}$, we must have

$$0 \quad j_w < t_w(\vec{t}):$$

Since $\vec{j}$ is the concatenation of $\vec{t}$ and j_w. So for a fixed value of $\vec{t}$, the number of possible $\vec{j}$ is $t_w(\vec{t})$. Thus, the total number of valid w-prefixes is bounded above by

$$\sum_{\vec{t} \in I_{w \ 1}(P)} t_w(\vec{t}) \quad |I_{w \ 1}(P)| \left(\max_{\vec{t} \in I_{w \ 1}(P)} t_w(\vec{t}) \right) \quad T_{w \ 1} \left(\max_{\vec{t} \in I_{w \ 1}(P)} t_w(\vec{t}) \right) = T_w$$

as claimed. This completes the induction, and proves the lemma.

To manage tuples of indices $\vec{\imath}$, it will be helpful to introduce some orders on these tuples.

Let $\vec{a}$ and $\vec{b}$ be tuples of integers, indexed by the same set I. Then $\vec{a} < \vec{b}$ if $a(i) < b(i)$ for all $i \in I$. In this case, we say $\vec{a}$ is less than $\vec{b}$ *componentwise*.

If $\vec{a}$ and $\vec{b}$ are both indexed by $[l]$ for some positive integer l, we write $\vec{a} \quad \vec{b}$ if there exists an index $i \in [l]$ such that $a_i < b_i$, and $a_j = b_j$ for all $j < i$. In this case we say that $\vec{a}$ is *lexicographically smaller* than $\vec{b}$.

Lemma 3.21. For any fixed integer $k \quad 1$ and constant $p \in (0; 1)$, there is a $(k; p)$-parameter family P consisting of sequences of constant positive integers, such that Algorithm 3 solves the kSAT-Prob$_p$ problem when given P as input.

Proof. We first describe the values of the parameters in P, and then prove that this choice of parameters enables Algorithm 3 to correctly solve kSAT-Prob$_p$.

We start by setting

$$t_1(?) = \lceil \log(2/p) \rceil: \tag{21}$$

The remaining parameters will be defined recursively.

For each valid index $\vec{\imath}$ of P, we will define $t_w(\vec{\imath})$ for $2 \quad w \quad k \quad 2$ in terms of the values of $s_v(\vec{\imath})$ for all $v < w$. For each valid index $\vec{\jmath}$ of P and $w \in [k \quad 2]$, we will define $s_w(\vec{\jmath})$ in terms of $t_w(\vec{\jmath})$, the values of $t_u(\vec{t})$ for all $u < w$ and valid indices $\vec{t}$, and the values of $s_v(\vec{\jmath})$ for all $v \in [k \quad 2]$ and valid indices $\vec{\imath} \quad \vec{\jmath}$. These dependencies will ensure that the parameters of P are well-defined. All these parameters will also depend on k and p, but by assumption these are constants.

For $w \quad 2$ and valid index $\vec{\imath}$, we set

$$t_w(\vec{\imath}) = \lceil \log_{2^w/(2^w \ 1)}(2/p) \rceil \left(w! \quad 2^w \prod_{v=0}^{w \ 1} s_v(\vec{\imath}) \right): \tag{22}$$

Algorithm 3. Threshold Satisfaction Algorithm for k-CNFs

Inputs: A k-CNF $'$, real $p \in (0;1)$, and $(k;p)$-parameter family $P = (t_w;s_w)_{w\in[k\ 2]}$
Returns: YES if $\Pr[\ '\ ] \geq p$, NO if $\Pr[\ '\ ] < p$

1. Initialize $\ '$, and $d \leftarrow d\log_{2^k=(2^k\ 1)}(2=p)e$.
   Initialize $i_w \leftarrow 0$ for each $w \in [k\ 2]$.
   Throughout the algorithm, we write $\vec{\imath} = \langle i_1;:::;i_{k\ 2}\rangle$ for convenience.
2. While $\vec{\imath} < \langle t_1(\vec{\imath});t_2(\vec{\imath});:::;t_{k\ 2}(\vec{\imath})\rangle$:
3. Compute $S \leftarrow \mathrm{Sun}_k\left(\ ;d;s_1(\vec{\imath});:::;s_{k\ 2}(\vec{\imath})\right)$.
4. If $D \leftarrow S$ is a disjoint set of size greater than d, return NO.
5. If instead $H \leftarrow S$ is a set of variables, compute

$$\Pr[\ ] = \sum_{:H\to\{0;1\}} \Pr[\ ]$$

   and return YES if this sum is at least p, NO if this sum is less than p.
6. Otherwise, S is a w-sunflower of size at least $s_w(\vec{\imath})$ for some $w \in [k\ 2]$.
   Set $C \leftarrow \mathrm{core}(\ )$. In this case:
7. If $w \geq 2$, then for each nonempty $B \subseteq C$:
8. Compute $_B \leftarrow \mathrm{MaxSun}_k(\ ;s_{jBj}(\vec{\imath});B)$.
9. If for some $B \subseteq C$, the call to MaxSun_k in step 8 returned a sunflower, let $B^?$ be the smallest such set, and run step 10:
10. Update $\ \leftarrow {}_{B^?}$.
    Update $C \leftarrow B^?$.
    Update $w \leftarrow jB^?j$.
11. Update $\ \leftarrow \mathrm{Assert}(\ ;C)$.
    Update $i_w \leftarrow (i_w + 1)$.
    Reset $i_y \leftarrow 0$ for all $w < y \leq k\ 2$.
12. Return NO.

From this equation, $t_w(\vec{\imath})$ depends only on p, w, and $s_v(\vec{\imath})$ for $v < w$.

For each valid index $\vec{\jmath}$ we also define the positive integer

$$m(\vec{\jmath}) = 2^{f_k(d;s_1(\vec{\jmath});:::;s_{k\ 2}(\vec{\jmath}))} \tag{23}$$

in terms of $s_w(\vec{j})$ for $w \in [k \quad 2]$, where f_k is the function from the statement of Proposition 3.11. Finally, we also set

$$(\vec{j}) = \min_{\vec{i} \quad \vec{j}} {}_{m(\vec{i})}(p) \tag{24}$$

where ${}_m(p)$, for any integer m, is the constant from the statement of Lemma 3.13, and the minimum is taken over all valid indices $\vec{i}$ which occur lexicographically after $\vec{j}$. From this equation, each $(\vec{j})$ depends only on p and the values of $s_w(\vec{i})$ over all $w \in [k \quad 2]$ and valid indices $\vec{i} \quad \vec{j}$.

For each $w \in [k \quad 2]$ and valid index $\vec{j}$ of P, we define $s_w(\vec{j})$ to be the smallest positive integer satisfying

$$s_w(\vec{j}) \quad 2w \left(\max_{v \in [w \quad 1]} s_v(\vec{j}) \right) \tag{25}$$

and

$$\left(1 \quad \frac{1}{2^{k \quad w}}\right)^{s_w(\vec{j})} \quad \frac{\min(p=2; \ (\vec{j}))}{(k \quad 2)T_{w \quad 1} t_w(\vec{j})} \tag{26}$$

where T_w is the parameter from Lemma 3.20. Note that $T_{w \quad 1}$ depends only on the values of $t_u(\vec{t})$ for $u < w$ and valid indices $\vec{t}$. thus, the above two equations define $s_w(\vec{j})$ in terms only of k, p, $t_w(\vec{j})$, the values of $t_u(\vec{t})$ for $u < w$ and valid indices $\vec{t}$, and the values of $s_v(\vec{i})$ for $v \in [k \quad 2]$ valid indices $\vec{i} \quad \vec{j}$.

Having defined the parameters of P, we now proceed with analyzing the algorithm.

When we first enter the loop of Algorithm 3, $\vec{i} = \vec{0}$ is the tuple consisting entirely of zeros, and $= '$ is the input formula.

The only way Algorithm 3 can execute an iteration of the loop without halting is if the call to Sun_k in step 3 of the algorithm returns a large sunflower. In this case, we run steps 6 through 11 of Algorithm 3. In step 11, we take the w-sunflower of size greater than $s_w(\vec{i})$ we found in and update

- by replacing it with a formula equivalent to $\wedge C$ for $C = \mathsf{core}( )$, and
- $\vec{i}$ by incrementing i_w by 1, and setting $i_y = 0$ for all $y > w$.

Since step 11 of Algorithm 3 is the only time in the algorithm the value of $\vec{i}$ changes, an easy induction argument with the second update rule above implies that each time $\vec{i}$ changes value in Algorithm 3, it strictly increases with respect to the lexicographic order.

B Claim 3.22 (Monoinvariant). The value of $\vec{i}$ is strictly increasing in the lexicographic order over the iterations of the loop from Algorithm 3.

Similarly, an easy induction argument using both the previously stated update rules proves the following claim about the behavior of Algorithm 3.

B Claim 3.23 (Loop Invariant). Whenever Algorithm 3 completes an iteration of its loop,

1. the formula is equivalent to

$$' \wedge \left(\bigwedge_{l=1}^{r} C_l \right)$$

where $C_1, \ldots, C_r$ are the cores of the sunflowers asserted in step 11 from previous iterations of the loop, and

2. for each $w \in [k-2]$, i_w is equal to the number of w-sunflowers found in previous iterations of the loop since the last time we found a v-sunflower for some $v < w$.

We can use these invariants to prove a more precise version of Claim 3.22, which will help us show that cores of sunflowers asserted in different iterations have size bounds which come from different parameters.

In what follows, if an execution of the loop in Algorithm 3 starts value with $i = j$, we refer to that iteration of the loop as *iteration* j. Note that by the loop condition from step 2 of Algorithm 3, any such j must be a valid index.

Claim 3.24 (Increasing Prefixes). Let $w \in [k-2]$. Let C_1 and C_2 be cores of sunflowers asserted in iterations j_1 and j_2 of the loop of Algorithm 3, for some $j_1 < j_2$. If C_1 was asserted as the core of a w-sunflower, then

$$(j_1)_1, \ldots, (j_1)_w) \prec (j_2)_1, \ldots, (j_2)_w).$$

Proof. Consider iteration j_1 of the loop. Let t denote the value of i at the time this iteration ends. Since clause C_1 is asserted in this iteration, we run step 11 of Algorithm 3 in this iteration. Since C_1 is a w-sunflower, this means that t satisfies

$$l_v = \begin{cases} (j_1)_v & \text{if } v < w \\ (j_1)_w + 1 & \text{if } v = w \\ 0 & \text{if } v > w. \end{cases} \tag{27}$$

By Claim 3.22, the value of i in all iterations after iteration j_1 will be greater than or equal to t in lexicographic order.

Take any iteration i occurring after iteration j_1. By assumption, $t \le i$.

If $t = i$, then by eq. (27) we have

$$(j_1)_1, \ldots, (j_1)_w) \prec (i_1, \ldots, i_w). \tag{28}$$

If instead $t < i$, there exist a smallest positive integer r with $l_r < i_r$. By eq. (27), we know that $l_v = 0$ for all $v > w$, so we must have $r \le w$. Then by eq. (27), we deduce taht Equation (28) holds in this case too.

So for any iteration i occurring after iteration j, eq. (28) holds. Specializing eq. (28) to the case of $i = j_2$ implies the desired result.

Suppose we just completed an iteration of the loop in Algorithm 2. Let $C_1, \ldots, C_r$ be all the cores of sunflowers we found and asserted in previous iterations of the loop, in the order they were found. Then by item 1 of Claim 3.23, φ is currently equivalent to

$$\varphi' \wedge \left(\bigwedge_{l=1}^{r} C_r \right). \tag{29}$$

For each $l \in [r]$, let define the function

$$\varphi_l = \varphi' \wedge \left(\bigwedge_{a=1}^{l-1} C_a \right) \wedge \neg C_l.$$

By definition, an assignment x satisfies $\quad_l$ precisely when x is a satisfying assignment of $\quad'$ with the property that l is the smallest positive integer such that C_l is not satisfied. In comparison, since $\quad$ is equivalent to the formula from eq. (29), the solutions to $\quad$ are precisely the satisfying assignments of $\quad'$ which also satisfy C_l for all $l \in [r]$. Since any assignment which does not satisfy all the C_l must have a unique earliest clause which is not satisfied, we deduce that

$$\Pr[\ '] \quad \Pr[\ ] = \sum_{l=1}^{r} \Pr[\ _l]. \tag{30}$$

For each $l \in [r]$, let j_l be the valid index such that C_l was asserted in iteration j_l of the loop. Let $w(l) = |C_l|$. Then C_l was the core of a $w(l)$-sunflower of size greater than $s_w(j_l)$ in a formula equivalent to

$$' \wedge \left(\bigwedge_{a=1}^{l \quad 1} C_a \right)$$

by item 1 of Claim 3.23. So by Proposition 3.12 we have

$$\Pr[\ _l] < \left(1 \quad \frac{1}{2^{k \quad w(l)}}\right)^{s_{w(l)}(j_l)}$$

for each $l \in [r]$.

Substituting the above inequality into eq. (30) yields

$$\Pr[\ '] \quad \Pr[\ ] \quad \sum_{l=1}^{r} \left(1 \quad \frac{1}{2^{k \quad w(l)}}\right)^{s_{w(l)}(j_l)}. \tag{31}$$

Our goal is to upper bound the sum from the right hand side of the inequality above. To do this, we define various collections of indices, to help group terms in this sum.

For each $w \in [k \quad 2]$, let

$$L_w = \{l \in [r] \mid w(l) = w\}$$

be the set of indices l for which C_l is the core of a w-sunflower.

For each valid $(w \quad 1)$-prefix $\hbar$, let $L_w(\hbar) \quad L_w$ be the set of indices l in L_w with

$$\hbar(j_l)_1; : : : ; (j_l)_{w \ 1}i = \hbar.$$

By construction, the $L_w(\hbar)$ partition $[r]$ as w ranges over $[k \quad 2]$ and $\hbar$ ranges over $I_{w \ 1}(P)$.

By splitting the sum in the right hand side of eq. (31) according to these parts, we have

$$\Pr[\ '] \quad \Pr[\ ] \quad \sum_{w=1}^{k \quad 2} \sum_{\hbar \in I_{w \ 1}(P)} \sum_{l \in L_w(\hbar)} \left(1 \quad \frac{1}{2^{k \quad w(l)}}\right)^{s_{w(l)}(j_l)}.$$

By eq. (26) and the above inequality, we get that

$$\Pr[\ '] \quad \Pr[\ ] \quad \sum_{w=1}^{k \quad 2} \sum_{\hbar \in I_{w \ 1}(P)} \sum_{l \in L_w(\hbar)} \frac{\min(p=2; \ (j_l))}{(k \quad 2)T_{w \ 1} t_w(j_l)} \tag{32}$$

By the definition of a parameter family in Definition 3.18, each integer $t_w(\vec{\jmath})$ depends only on the first $w - 1$ entries of $\vec{\jmath}$. For $l \in L_w(\hbar)$ we have

$$h(j_l)_1, \ldots, (j_l)_{w-1}i = \hbar \tag{33}$$

so we have $t_w(\vec{\jmath}_l) = t_w(\hbar)$ in this case. Making this substitution in eq. (32) gives

$$\Pr[\vec{\imath}'] \leq \Pr[\vec{\imath}] \sum_{w=1}^{k-2} \sum_{\hbar \in I_{w-1}(P)} \sum_{l \in L_w(\hbar)} \frac{\min(p-2, \ell(\vec{\jmath}_l))}{(k-2)T_{w-1} t_w(\hbar)} \tag{34}$$

By Claim 3.24, we know that the w-tuples

$$h(j_l)_1, \ldots, (j_l)_w i$$

are different for each $l \in L_w$. Then by eq. (33), the values of the $(j_l)_w$ are pairwise distinct for the different choices of $l \in L_w(\hbar)$. Thus eq. (19) implies that for any $\hbar$ there are at most

$$|L_w(\hbar)| \leq t_w(\hbar)$$

choices for valid w-prefixes $\vec{\jmath}$ which agree with $\hbar$ in the first $w-1$ coordinates. This discussion implies that for any fixed choice of $w \in [k-1]$ and $\hbar \in I_{w-1}(P)$, we can bound

$$\sum_{l \in L_w(\hbar)} \frac{\min(p-2, \ell(\vec{\jmath}_l))}{(k-2)T_{w-1} t_w(\hbar)} \leq \max_{l \in L_w(\hbar)} \left[\frac{\min(p-2, \ell(\vec{\jmath}_l))}{(k-2)T_{w-1}} \right]$$

because there are at most $t_w(\hbar)$ terms in the sum.

Since iteration $\vec{\jmath}_l$ occurs before iteration $\vec{\imath}$ for every $l \in [r]$, we have $\vec{\imath} \not\preceq \vec{\jmath}_l$ for all $l \in [r]$ by Claim 3.22. From eq. (24), we therefore have $\ell(\vec{\jmath}_l) \leq m(\vec{\imath})(p)$ for every $l \in [r]$. Substituting this bound into the above inequality implies that

$$\sum_{l \in L_w(\hbar)} \frac{\min(p-2, \ell(\vec{\jmath}_l))}{(k-2)T_{w-1} t_w(\hbar)} \leq \frac{\min\left(p-2, m(\vec{\imath})(p)\right)}{(k-2)T_{w-1}}.$$

By summing over all choices of w and $\hbar$ in the above inequality and combining with eq. (34), we get that

$$\Pr[\vec{\imath}'] \leq \Pr[\vec{\imath}] \sum_{w=1}^{k-2} \sum_{\hbar \in I_{w-1}(P)} \left(\frac{1}{(k-2)T_{w-1}} \min\left(p-2, m(\vec{\imath})(p)\right) \right).$$

By Lemma 3.20, the set $I_{w-1}(P)$ contains at most T_{w-1} elements. So for each $w \in [k-2]$, the inner summation on the right hand side of the above inequality involves at most T_{w-1} summands. Thus we have

$$\Pr[\vec{\imath}'] \leq \Pr[\vec{\imath}] \sum_{w=1}^{k-2} \left(\frac{1}{k-2} \min\left(p-2, m(\vec{\imath})(p)\right) \right) \leq \min\left(p-2, m(\vec{\imath})(p)\right). \tag{35}$$

Having upper bounded the difference between the satisfaction probability of and the input formula, we can now show correctness of Algorithm 3. We consider cases based off whether Algorithm 3 halts within its loop (at steps 4 or 5) or outside of its loop (at step 12).

Case 1: Halting Within the Loop

Suppose Algorithm 3 halts on iteration i of the loop.

If we halt in step 4 of Algorithm 3, has a disjoint set of size greater than d. Then by Proposition 3.3 we have

$$\Pr[\ ] < \left(1 \quad \frac{1}{2^k}\right)^d \quad p=2$$

since $d > \log_{2^k=(2^k \ 1)}(2=p)$.

By eq. (35) we have

$$\Pr[\ '] \quad \Pr[\ ] \quad p=2:$$

Adding these two inequalities together, we get that

$$\Pr[\ '] = \Pr[\ ] + (\Pr[\ '] \quad \Pr[\ ]) < p=2 + p=2 = p$$

so Algorithm 3 correctly returns NO in this case.

The other possibility is that we halt in step 5 of Algorithm 3. In this case, by Proposition 3.11, we find a set H of at most $f_k(d; s_1(i); : : : ; s_{k \ 2}(i))$ variables in with the property that for every assignment $: H \quad f0; 1g$, the induced formula is a 1-CNF.

By Proposition 2.19 we have

$$\Pr[\ ] = \sum_{: H \ \square \ f0;1g} \Pr[\quad]:$$

By Proposition 2.11, each summand in the right hand side above is zero or a power of two. So the above equation shows that $\Pr[\ ]$ is a sum of at most

$$2^{jHj} \quad 2^{f_k(d;s_1(i);:::;s_{k \ 2}(i))} = m(i)$$

powers of two by eq. (23).

If $\Pr[\ ] \quad p$, then since is equivalent to the formula from eq. (29), we have

$$\Pr[\ '] \quad \Pr[\ ] \quad p$$

so Algorithm 2 returns YES correctly in this case.

If $\Pr[\ ] < p$, by Lemma 3.13 we have

$$\Pr[\ ] \quad p$$

for $= \ _{m(i)}(p)$.

By eq. (35) we have

$$\Pr[\ '] \quad \Pr[\ ] < \ :$$

Adding the previous two inequalities together yields

$$\Pr[\ '] = \Pr[\ ] + (\Pr[\ '] \quad \Pr[\ ]) < (p \quad) + \quad = p$$

so Algorithm 3 returns NO correctly in this case.

Thus Algorithm 3 returns the correct answer whenever it halts within the loop.

Case 2: Halting Outside the Loop

In this case, Algorithm 3 reaches step 12. By the loop condition in step 2 of Algorithm 3, this is only possible if there exists an index $w \in [k \quad 2]$ such that

$$i_w > t_w(i_1; : : : ; i_{w \ 1}) \tag{36}$$

when the algorithm halts.

By item 2 of Claim 3.23, i_w is equal to the number of w-clauses which have been asserted in since the last time a clause of width less than v has been asserted in . So there exists a collection B of i_w cores of w-sunflowers which were asserted in , with the property that for all $v < w$, any core of a v-clause asserted in must have been asserted in the iterations before any core in B was asserted. Let

$$= \bigwedge_{C \in B} C$$

be the conjunction of all clauses in B. The following claim about will help us reason about the satisfaction probability of at the time the algorithm halts.

B Claim 3.25 (Avoiding Large Sunflowers). The formula does not contain a v-sunflower of size greater than $s_v(i)$ for any $v \in [w \quad 1]$.

Proof. The result vacuously holds for $w = 1$, so we may assume that $w \quad 2$.

Suppose to the contrary there exists $v \in [w \quad 1]$ so that contains a v-sunflower of size $s_v(i) + 1$. Among all clauses in , let B be the last one which was asserted in . Let j be the iteration of the algorithm where B was asserted. At the beginning of iteration j, must have contained a w-sunflower (B) of size greater than $s_w(j)$.

We claim that at the beginning of iteration j, must have contained $n \ fBg$ as a subformula. Indeed, by the definition of B, every clause in $n \ fBg$ was asserted in before iteration j. When a clause C is asserted in , it is added as a clause. Moreover, by the property of B, no clauses of width less than w can have been asserted in after the clauses of $n \ fBg$ were asserted. This means that none of the clauses in $n \ fBg$ could have been removed from by other assertions (since when a clause C of width at least w is asserted, it only removes clauses containing C as a proper subset because $w \quad 2$, and in particular cannot remove any w-clauses).

Now, since $B \in B$, we know that no clause of width less than w can have been asserted in after iteration j, so by Claim 3.23 it must be the case that

$$hi_1; : : : ; i_{w \ 1}i = hj_1; : : : ; j_{w \ 1}i : \tag{37}$$

Since consists of w-clauses, the clauses of nB contain at most $ws_v(i)$ distinct variables. By eq. (25) and eq. (37), we also know that petals((B)) has greater than

$$s_w(j) > 2w \ \ s_v(j) = 2w \ \ s_v(i)$$

clauses. Since petals((B)) is a disjoint set, and every literal corresponds to at most two variables, we deduce that petals((B)) must contain some clause C which shares no variables in common with any clause of $\quad n\, fBg$.

Then $B \;[\; C$ is a clause of $\quad (B)$. The discussion in the previous paragraph implies that

$$^{0} = (\quad [\; fC \;[\; Bg) \, n \, fBg$$

is a v-sunflower of size $j \;\; j = s_v(\vec{\imath}\,) + 1$ with the same core as $\quad$. We already showed that every clause in $\quad$ except B belong to $\quad$ at the beginning of iteration $\vec{\jmath}$. Since $\quad$ contains (B) at the beginning of this iteration, it also has $B \;[\; C$ as a clause. So $\quad ^{0}$ is a v-sunflower of size greater than $s_v(\vec{\imath}\,)$ in $\quad$ at the beginning of iteration j. By eq. (33) and the fact that $s_v(\vec{\imath}\,) = s_v(hi_1; : : : ; i_v i)$, we have $s_v(\vec{\imath}\,) = s_v(\vec{\jmath}\,)$, so that $\quad$ in fact contains a v-sunflower $\quad ^{0}$ of size greater than $s_v(\vec{\jmath}\,)$ at the beginning of iteration $\vec{\jmath}$.

Moreover, core(^{0}) = core($\quad$) $\quad B$ since B is a clause of $\quad$.

We are ready to establish our contradiction. By definition of $\vec{\jmath}$, in iteration $\vec{\jmath}$ of the loop, we must set assert B in $\quad$. Either this happens because we find B as the core of a sunflower in step 6, or because we find B as the core of a sunflower in steps 7 and 8 of Algorithm 3.

If we find B as the core of a sunflower in step 6, then because $jBj = w \quad 2$, we will run steps 7 and 8 and, by the previous discussion, find, for some $v < w$, a v-sunflower of size greater than $s_v(\vec{\jmath}\,)$ in $\quad$ whose core is a subset of B. Then in steps 9 through 11 we will assert the core of this v-sunflower instead of B, which contradicts our choice of $\vec{\jmath}$.

If instead we find B as the core of a sunflower in steps 7 and 8, in these same steps the algorithm will also looked for v-sunflowers of size greater than $s_v(\vec{\jmath}\,)$ for each subset of B of size v, for all $v < w$. By the previous discussion, such a v-sunflower exists in $\quad$ in this iteration, and will be found. The condition in step 9 ensures we assert the core of a sunflower of minimum weight among all those we found in steps 7 and 8, so in this case too B would not be asserted, which again yields a contradiction.

In each case we get a contradiction, so our initial assumption was false, and $\quad$ does not contain a v-sunflower of size greater than $s_v(\vec{\imath}\,)$ for any $v \; 2 \; [w \quad 1]$. □

Using Claim 3.25, we deduce the following.

B Claim 3.26. We have Pr[ ] < p=2.

Proof. From eqs. (22) and (36) we know that $\quad$ is a w-CNF with more than

$$d\log_{2^w=(2^w \quad 1)}(2=p)e \quad \left(w! \quad 2^w \prod_{v=0}^{w \;\; 1} s_v(\vec{\imath}\,) \right)$$

clauses.

Then by Proposition 3.17, $\quad$ either contains a disjoint set of size greater than

$$d\log_{2^w=(2^w \quad 1)}(2=p)e$$

or contains a v-sunflower of size greater than $s_v(\vec{\imath}\,)$ for some $v \; 2 \; [w \quad 1]$. By Claim 3.25 this latter case cannot occur, so $\quad$ has a disjoint set of size more than

$$d = d\log_{2^w=(2^w \quad 1)}(2=p)e:$$

Consequently, Proposition 3.3 implies that

$$\Pr[\ ] < \left(1 \quad \frac{1}{2^w}\right)^d \quad p=2$$

as claimed. □

At the time Algorithm 3 halts, we have , so by Claim 3.26 we have

$$\Pr[\ ] \quad \Pr[\ ] < p=2:$$

By Equation (35) we have

$$\Pr[\ '] \quad \Pr[\ ] < p=2:$$

Adding these two inequalities together yields

$$\Pr[\ '] = \Pr[\ ] + (\Pr[\ '] \quad \Pr[\ ]) < p$$

so Algorithm 3 correctly returns NO in this case.

So in both case 1 and case 2, Algorithm 3 solves kSAT-Prob $_p$ correctly as claimed.

Proof of Theorem 2.15. By Lemma 3.21, for any fixed integer k 1 and choice of constant p 2 $(0;1)$, there exists a $(k;p)$-parameter family $P = (t_i;s_i)_{i2[k\ 2]}$ consisting of constant positive integers such that Algorithm 3 correctly solves kSAT-Prob $_p$ when given the parameters in P as input. So to prove the theorem, it suffices to show that Algorithm 3 runs in linear time when given these constant parameters.

Step 1 of Algorithm 3 runs in linear time since we just read the input.

The condition in the while loop of Algorithm 3 ensures that the algorithm only executes the loop when $\vec{\imath}$ is a valid index for P. Moreover, by Claim 3.22 every iteration of the loop in Algorithm 3 begins with a different value of $\vec{\imath}$. Any valid index is a valid $(k \quad 2)$-prefix, so by Lemma 3.20 the number of valid indices for P (and thus the number of iterations of the loop in Algorithm 3) is at most $T_{k\ 2}$, which is constant since the parameters in P are constants. So steps 3 through 11 of Algorithm 3 are each executed at most a constant number of times.

Step 3 of Algorithm 3 takes linear time by Proposition 3.11 because the $s_w(\vec{\imath})$ are constants and $d = d\log_{2^k=(2^k\ 1)}(2=p)e$ is also a constant for constant k and p.

Step 4 of Algorithm 3 takes linear time since we just need to read a disjoint set for .

Step 5 of Algorithm 3 takes at most $2^{jH}j\ 'j$ time asymptotically. By Proposition 3.11 the number of variables in H is bounded above by a constant since k, d, and the $s_w(\vec{\imath})$ are constants for all w 2 $[k \quad 2]$, so this step takes linear time.

Step 6 of Algorithm 3 takes linear time since we just need to read a sunflower for .

Steps 7 and 8 of Algorithm 3 involve making at most 2^k calls to MaxSun_k with size parameters of the form $s_w(\vec{\imath})$. Since the $s_w(\vec{\imath})$ are constants, by Corollary 3.16 each call to MaxSun_k runs in linear time. Since k is a constant, these steps overall run in linear time.

Steps 9 and 10 of Algorithm 3 involve reading the at most 2^k sunflowers returned in steps 8 and 10, which takes linear time since k is constant.

Step 11 of Algorithm 3 takes linear time by Proposition 2.20.

Step 12 of Algorithm 3 takes constant time since we just return NO.

So overall Algorithm 3 takes linear time, as claimed.

3.4 Commentary on Algorithms

Exact Parameterized Complexity

How does the runtime of our kSAT-Prob$_p$ algorithm depend on the parameters k and p? Investigating the exact time complexity of algorithms for this problem is important if we care about solving kSAT-Prob$_p$ in practice.

For $k = 2$, the proof of Theorem 3.1 shows that given a 2-CNF $'$ and real $p \in (0;1)$, we can determine if $\Pr[\,'\,] \geq p$ in

$$2^{2b\log_{4=3}(1=p)c}j\,'\,j = \mathrm{poly}(1=p)j\,'\,j$$

time asymptotically. This $\mathrm{poly}(1=p)$ dependence in the runtime seems reasonable, since we can solve #2SAT by by solving 2SAT-Prob$_p$ for $n+1$ different values of p in a binary search argument (as sketched in the discussion after the proof of Corollary 2.10). since #2SAT is #P-hard, we do not expect this problem to be solvable in polynomial time, so it makes sense that 2SAT-Prob$_p$ should also not be polynomial-time solvable in the regime where $(1=p) \geq 2^{(n)}$ for example.

For $k \geq 3$, the runtime dependence on p in our algorithm for kSAT-Prob$_p$ is much more chaotic. Because our algorithms in this case involve parameters based off the binary gap value $m(p)$ from Lemma 3.13, their runtime is based not just on the magnitude of p, but in general depend on the binary representation of p.

Nonetheless, to get some sense of the parameterized complexity of our algorithms in this regime, we analyze the time complexity of our kSAT-Prob$_p$ algorithm for the special case where $k = 3$ and $p = (1=2)^a$ for some positive integer a.

Corollary 3.27. Let m be a positive integer. If $p = (1=2)^a$ for some positive integer a, then we can take $m(p) = (1=2)^{a+m}$ in Lemma 3.13.

Proof. By the formula for an infinite geometric series, we can write

$$p = \frac{1}{2^a} = \sum_{i=1}^{1} \frac{1}{2^{a+i}}:$$

By eq. (11) from the proof of Lemma 3.13, we can take

$$m(p) = \sum_{i=m+1}^{1} \frac{1}{2^{a+i}} = \frac{1}{2^{a+m}}$$

using the formula for an infinite geometric series again, as claimed.

To solve 3SAT-Prob$_p$ in linear time for constant p, we set the input parameters $s(i)$ in Algorithm 2 according to eqs. (13) and (14) from the proof of Lemma 3.14 via

$$s(t) = d\log_{4=3}(2t=p)e \tag{38}$$

and recursively setting

$$s(i) = \left\lceil \log_{4=3}\left(\frac{i}{m_{i+1}(p)}\right)\right\rceil \tag{39}$$

for each $i \in [t-1]$, where $m_{i+1}(p)$ is defined in eq. (12) as

$$m_{i+1} = 2^{f_3(d;s(i+1))}. \tag{40}$$

We now derive concrete bounds for these parameters in the special case of $p = (1/2)^a$, to get a sense of how quickly (or perhaps more accurately, how *slowly*) Algorithm 2 runs in terms of p.

In our special case of $p = (1/2)^a$, by Corollary 3.27 and eq. (39) we can write

$$s(i) = \left\lceil \log_{4/3}\left(\frac{i}{m_{i+1}(p)}\right)\right\rceil = \left\lceil \log_{4/3}\left(2^{m_{i+1}+p} \cdot i\right)\right\rceil = O(m_{i+1} + p + \log i). \tag{41}$$

From eqs. (5) and (6) in the proof of Proposition 3.11, we have

$$f_3(d; s(i+1)) \cdot 2^{3d} \cdot f_2((1+3d) \cdot s(i+1)) = 2^{3d+2(1+3d)\,s(i+1)} = 2^{O(d\,s(i+1))}.$$

Combining the above inequality with eq. (40) yields

$$m_{i+1} = 2^{2^{O(d\,s(i+1))}}.$$

Substituting the above bound into eq. (41) yields

$$s(i) = 2^{2^{O(d\,s(i+1))}} + O(p + \log i) = 2^{O(1/p)^{O(s(i+1)}} + O(p + \log i).$$

where we used the fact that $d = O(\log_{8/7}(2/p))$ in Algorithm 2.

Starting with the value of $s(t)$ from eq. (38) and applying the recurrence from the above bound repeatedly, we deduce we can bound $s(1)$ above by an exponential tower of height $O(t)$, where each term in tower is at most $O(1/p)$. The asymptotic runtime of Algorithm 2 is bounded above by the call to Sun_3 in step 3 of the algorithm, with parameters d and $s(1)$, which runs in time exponential in $s(1)$. So overall, the asymptotic runtime of 3SAT-Prob_p grows like an exponential tower of $O(1/p)$ terms, of height $O(\log(1/p))$.

Regularity

As explored in the Exact Parameterized Complexity subsection of Section 3.4, the runtime of our $k\text{SAT-Prob}_p$ algorithms can increase at a dramatic pace as p gets smaller, even in the special case where $k = 3$ and p is constrained to be a power of two, In particular, the runtime can have a tower-of-exponents dependence on $1/p$.

In retrospect however, this atrocious runtime dependence on p is perhaps not too surprising, given the similarities between our approach to solving $k\text{SAT-Prob}_p$ and various *regularity lemma* arguments in the mathematics and computer science literature.

At a high level, given a class C of combinatorial objects, a regularity lemma for that class is a structural theorem which states that, given any object $A \in C$ and constant $p > 0$ (think of p as as some sort of *robustness* or *error* parameter), after a small number of edits, A admits some sort of structured representation of "small size." What makes a regularity lemma interesting is that the "small number" of edits and "small size" bound on the structured

representation both depend only on p. In particular, the size bounds in a regularity lemma *do not depend on the size of* C.

The most famous example of this phenomenon is Szemerédi's classic graph regularity lemma [Sze75], which takes C to be the class of undirected graphs. In standard proofs of Szemerédi's regularity lemma, the size of the structured representation obtained for a graph grows like an exponential tower of twos whose height is polynomial in $1/p$, where $p > 0$ is an error parameter. Moreover, there are lower bounds showing that this exponential tower dependence is *necessary* in the graph regularity lemma [Gow97, MS14].

Our results for kSAT-Prob$_p$ can also be viewed as proving a regularity lemma, for the class C of all k-CNF formulas for some fixed integer k. In particular, the proof of Lemma 3.21 shows that for any $p \in (0;1)$, there exists a family of constant parameters P depending only on p such that Algorithm 3 solves kSAT-Prob$_p$ when given P as input, and the proof of Theorem 2.15 shows that in this case, the loop of Algorithm 3 executes at most a constant depending on p number of times before halting. Following the steps of Algorithm 3, this implies that, given a fixed integer k 1, for any k-CNF formula $'$ and $p \in (0;1)$, either

- we have Pr[$'$] < p (intuitively, the formula $'$ is *random-like*),
- or instead Pr[$'$] p (intuitively, the formula $'$ is *highly-structured*), and we can repeatedly assert cores of sunflowers in $'$ to produce a new formula which also has Pr[ ] p, and whose solution space can be written as the disjoint union of the satisfying assignments for some induced 1-CNF formulas . The number of cores we assert (the "small number of edits") and the number of induced 1-CNFs in the final decomposition (the "small-sized" structured representation) both depend only on p.

We formally record this regularity lemma below. To help state this result, we say that a formula is obtained by asserting t clauses in a CNF formula $'$, if there exists a sequence of formulas $'_0;\dots;'_t$ and clauses $C_1;\dots;C_t$ such that $' = '_0$, we have

$$'_i = \mathsf{Assert}('_{i-1};C_i)$$

for each $i \in [t]$, and $= '_t$.

Theorem 3.28: Threshold Satisfaction Regularity for k-CNFs

For every integer k 1 and real $p \in (0;1)$, there are integers $h;t$ 1 such that for all k-CNF formulas $'$ with Pr[$'$] p, there exists a k-CNF formula and subset H of at most h variables of , such that

1. we have Pr[ ] p,
2. the formula is obtained by taking $'$ and asserting at most t clauses, and
3. for all assignments $: H$ $f0;1g$, the induced formula is a 1-CNF.

Proof. Take an arbitrary k-CNF formula $'$ satisfying Pr[$'$] p. By the proof of Lemma 3.21, there is a $(k;p)$-parameter family $P = (t_1;s_1;\dots;t_{k-2};s_{k-2})$ whose parameters depend only on k and p, such that Algorithm 3 solves kSAT-Prob$_p$ when given P as input.

Run Algorithm 3 on $'$ with the parameters in P. Since $\Pr[\,'\,] \quad p$, this call to Algorithm 3 returns YES. The only place that Algorithm 3 can return YES in is step 5. Let $\vec{\imath}$ denote the value of $\langle i_1;\ldots;i_{k\ 2}\rangle$ at the time Algorithm 3 halts. Let denote the formula from Algorithm 3 at the time the algorithm halts. Since the algorithm returns YES, we must have $\Pr[\ ] \quad p$, so condition 1 from the theorem holds.

Step 5 of Algorithm 3 can only be reached if the identified a set of variables H in in the call to Sun_k in step 3 of Algorithm 3. By definition, H is the output of

$$\mathsf{Sun}_k\left(\ ;d;s_1(\vec{\imath});\ldots;s_{k\ 2}(\vec{\imath})\right)$$

in step 3 of Algorithm 3, where $d = d\log_{2^k=(2^{k\ 1})}(2=p)e$. Then by condition 2 of Proposition 3.11 we know that for every $: H \quad f0;1g$, the induced formula is a 1-CNF, so condition 3 from the theorem holds.

By assumption, the parameters $s_w(\vec{\imath})$ for $w \in [k \quad 2]$ from P are bounded above by a function of p and k. By condition 2 of Proposition 3.11 again, we know that

$$jHj \quad f_k(d;s_1(\vec{\imath});\ldots;s_{k\ 2}(\vec{\imath})) \tag{42}$$

for some function f_k. Now, since Algorithm 3 halts in step 5 within the loop, $\vec{\imath}$ must satisfy the inequality from step 2 of Algorithm 3. In particular, $\vec{\imath}$ is a valid prefix for P. Define

$$h = h(k;p) = \max_{\vec{\jmath}\in I(P)} f_k(d;s_1(\vec{\jmath});\ldots;s_{k\ 2}(\vec{\jmath})):$$

Since the parameters in P depend only on k and p, h is indeed a function of k and p. Since $\vec{\imath}$ is a valid index for P, by eq. (42) we have $jHj \quad h$.

By Claim 3.23, is obtained by taking $'$ and asserting clauses found in step 11 of each iteration of the loop in Algorithm 3 before halting. By the inequality in step 2 of Algorithm 3, the tuple $\langle i_1;\ldots;i_{k\ 2}\rangle$ is a valid index for P at the beginning of iteration of loop. By Claim 3.22, these tuples distinct in each iteration of the loop. Let

$$t = t(k;p) = T_{k\ 2}$$

be the upper bound on the number of valid $(k \quad 2)$-prefixes, or equivalently valid indices, for P defined in Lemma 3.20. By Lemma 3.20, the value of $T_{k\ 2}$ depends only on the values of parameters in P. Since the parameters of P depend only on k and p, the integer t is indeed a function of k and p. By the previous discussion, our call to Algorithm 3 executes at most t iterations of the loop. Thus is obtained by asserting at most t clauses in $'$, so condition 2 from the theorem holds.

Since we have shown that conditions 1 through 3 from the theorem statement are satisfied by our choices of h and t, the desired result holds.

Chapter 4

Variants of Threshold Satisfiability

In this chapter, we explore generalizations and variations of the kSAT-Prob$_p$ problem, probing the limits of tractability for threshold satisfaction. In some cases, we will find that our kSAT-Prob$_p$ algorithms can be generalized to solve interesting related tasks. In other situations, we will see that the tractability of kSAT-Prob$_p$ is surprisingly brittle, and even minor modifications to the problem definition can produce seemingly hard problems.

4.1 Strict Thresholds

In Chapter 3 we saw algorithms for determining if the satisfaction probability of a k-CNF formula is at least p for some constant $p \in (0, 1)$. What if instead we want to determine if a formula has satisfaction probability *strictly greater than* p? To explore this question, we introduce GtMajority-SAT, a strict threshold variant of Majority-SAT.

GtMajority-SAT
Given a CNF formula φ, determine if $\Pr[\varphi] > 1/2$.

Analogously, we introduce kSAT-Prob$_{>p}$, a strict threshold variant of kSAT-Prob$_p$.

kSAT-Prob$_{>p}$
Given a k-CNF formula φ, determine if $\Pr[\varphi] > p$.

It is known that GtMajority-SAT is PP-complete, just like the Majority-SAT problem.

Proposition 4.1. GtMajority-SAT is PP-complete.

Proof. We prove the result by showing that GtMajority-SAT is in PP, and is PP-hard.

Claim 4.2. GtMajority-SAT is in PP.

Proof. We give a polynomial-time reduction from GtMajority-SAT to Majority-SAT. This will prove the claim, since Majority-SAT $\in$ PP.

Take an arbitrary instance of GtMajority-SAT, consisting of a CNF formula $'$ on n variables. Introduce variables $y_1; : : : ; y_n$ not in $'$. Construct the CNF formula

$$= ' \wedge (y_1 \vee \quad \vee y_n)$$

on $2n$ variables. We can construct in linear time given $'$.

Since the y_i variables do not appear in $'$, we have

$$\Pr[\ ] = \Pr[y_1 \vee \quad \vee y_n] \quad \Pr['] = (1 \quad (1=2)^n) \Pr[']: \tag{43}$$

If $\Pr[']$ 1=2, then by eq. (43) we have

$$\Pr[\ ] \quad (1 \quad (1=2)^n) \quad (1=2) < 1=2:$$

If instead $\Pr['] > 1=2$, because $'$ has n variables, we have $\Pr[']$ $(1=2) + (1=2)^n$. Combining this inequality with eq. (43) implies that

$$\Pr[\ ] \quad (1 \quad (1=2)^n)\,((1=2) + (1=2)^n) = 1=2 + (1=2)^{n+1} \quad (1=2)^{2n} \quad 1=2:$$

Thus $\Pr['] > 1=2$ if and only if $\Pr[\ ]$ 1=2, so the transformation from $'$ to is a correct reduction from GtMajority-SAT to Majority-SAT. By the discussion in the first paragraph of this proof, this shows the claim. □

B Claim 4.3. GtMajority-SAT is PP-hard.

Proof. We give a polynomial-time reduction from Majority-SAT to GtMajority-SAT. This will prove the claim, since Majority-SAT is PP-hard.

Take an arbitrary instance of Majority-SAT, consisting of a CNF formula $'$ over the n variables $x_1; : : : ; x_n$. Let be the CNF formula

$$= x_1 \wedge \left(\bigwedge_{i=2}^{n} (: x_1 \vee x_i) \right) :$$

Note that an assignment satisfies if and only if x_1 is set true, or x_1 is false and x_i is set to true for all i 2. Thus

$$\Pr[\ ] = (1=2) + (1=2)^n: \tag{44}$$

Introduce a new variable y. Construct the CNF formula

$$= (y \vee ') \wedge (: y \vee \ )$$

on $n + 1$ variables. We can construct in linear time given $'$.

Every assignment must set y to be either or true or false, so we have

$$\Pr[\ ] = \Pr[\ \wedge y] + \Pr[\ \wedge : y]:$$

The formula $(\wedge y)$ is equivalent to $(\wedge y)$. Similarly, the formula $(\wedge : y)$ is equivalent to $(' \wedge : y)$. Since the variable y does not appear in $'$ or , we have

$$\Pr[' \wedge : y] = (1=2) \quad \Pr[']$$

and

$$\Pr[\ \wedge y] = (1/2)\ \Pr[\ ]$$

so

$$\Pr[\ ] = (1/2)\ (\Pr[\ '] + \Pr[\ ]) = (1/2)\ (\Pr[\ '] + (1/2) + (1/2)^n) \tag{45}$$

where we used the value of $\Pr[\ ]$ from eq. (44).

If $\Pr[\ ']$ $1/2$, then by eq. (45) we have

$$\Pr[\ ]\ (1/2)\ ((1/2) + (1/2) + (1/2)^n) > 1/2:$$

If instead $\Pr[\ '] < 1/2$, then in fact $\Pr[\ ']$ $(1/2)$ $(1/2)^n$ because $'$ has n variables. So by eq. (45) we have

$$\Pr[\ ]\ (1/2)\ ((1/2) + (1/2)^n + (1/2)\ (1/2)^n) = 1/2:$$

Thus $\Pr[\ ']$ $1/2$ if and only if $\Pr[\ ] > 1/2$, so the transformation from $'$ to is a correct reduction from Majority-SAT to GtMajority-SAT. By the discussion in the first paragraph of this proof, this shows the claim. □

Combining Claims 4.2 and 4.3 proves the desired result.

So the complexity of GtMajority-SAT and Majority-SAT are the same. However, the reduction from GtMajority-SAT from Majority-SAT presented in the proof of Claim 4.2 involves using a clause of width n, and so we cannot use this argument to reduce kSAT-Prob$_{>1/2}$ to kSAT-Prob $_{1/2}$ for constant k. So how difficult is kSAT-Prob$_{>p}$ for constant k? Does this problem have the same complexity as kSAT-Prob $_p$, or does kSAT-Prob$_{>p}$ remain as hard as the GtMajority-SAT problem?

Perhaps surprisingly, neither of these possibilities is the correct answer.

Theorem 4.4: Strict k-CNF Thresholds at 1/2

For positive integers k $O(1)$, the kSAT-Prob$_{>1/2}$ problem is

- polynomial-time solvable for k 3, and
- NP-complete for k 4.

Theorem 4.4 is interesting because for constants k 4, it shows that the kSAT-Prob$_{>p}$ problem is much harder than the closely related kSAT-Prob $_p$ problem, yet much easier than the general Majority-SAT problem, even though this problem on general CNF formulas is *equivalent* to the corresponding GtMajority-SAT problem by Proposition 4.1.

We prove each of the parts of Theorem 4.4 separately in individual lemmas below. We begin with the hardness result. The basic construction used in this hardness reduction (and other reductions later on) is that, given a formula $'$ and a variable x it does not contain, the formula

$$= fC\ [\ fxg\ j\ C\ 2\ 'g$$

obtained by adding x to each clause of $'$ is always satisfied by assignments that set x true, and has additional solutions if and only if $'$ is satisfiable.

Idea 6 Adding new variables to every clause of a CNF formula shifts its fraction of satisfying assignments and increases its clause widths in a predictable way.

Lemma 4.5. For any fixed integer $k \quad 4$, kSAT-Prob$_{>1=2}$ is NP-hard.

Proof. Let $j = k \quad 1$. Since $k \quad 4$, the jSAT problem is NP-hard. So to prove the result, it suffices to show a polynomial-time reduction from jSAT to kSAT-Prob$_{>1=2}$.

Let $'$ be an arbitrary j-CNF. Let v be a variable not in $'$. Let $\quad$ be the formula

$$= fC \; [\; fvg \; j \; C \; 2 \; 'g$$

obtained by adding v to every clause in $'$. Given $'$, we can construct $\quad$ in linear time.

Since every satisfying assignment of $\quad$ sets v to be either true or false, we have

$$\Pr[\quad] = \Pr[\quad \wedge v] + \Pr[\quad \wedge :v]: \tag{46}$$

Since every clause of $\quad$ contains v, we have

$$\Pr[\quad \wedge v] = \Pr[v] = 1=2:$$

If we delete v from each clause of $\quad$ we recover $'$, so

$$\Pr[\quad \wedge :v] = (1=2) \quad \Pr[\; ']$$

where we are viewing $\quad$ as a formula on one more variable than $'$.

Substituting these equations back into eq. (46) yields

$$\Pr[\quad] = (1=2) \quad (1 + \Pr[\; ']) :$$

The above equation shows that $\Pr[\quad] > 1=2$ if and only if $'$ is satisfiable.

Thus the map from $'$ to $\quad$ is a valid reduction from jSAT to kSAT-Prob$_{>1=2}$.

By the first paragraph of this proof, this proves the lemma.

Next, we show how to place kSAT-Prob$_{>1=2}$ in the class NP, for constant k. That is, given a k-CNF $'$ for constant k, if $\Pr[\; '] > 1=2$ there is always short certificate which can convince us of this fact in polynomial time.

Why should this be possible? Well, Theorem 2.15 already shows that in linear time we can determine if $\Pr[\; '] \quad p$, without the need for any certificate. If this inequality does not hold, then certainly $\Pr[\; '] > p$ does not hold either. So we can focus on the case where we already know, from using our algorithm for kSAT-Prob $_p$, that $\Pr[\; '] \quad p$.

Idea 7 If we know that $\Pr[\; '] \quad p$, then a certificate for the existence of *one more* satisfying assignment beyond the p-fraction of assignments which we already know are satisfying should help us determine if $\Pr[\; '] > p$.

Algorithm 3 solves kSAT-Prob $_p$ by *inferring* a formula $\quad$ whose solutions are all satisfying assignments of $'$, with the property that $\Pr[\; '] \quad p$ if and only if $\Pr[\quad] \quad p$. Given this information, we can implement Idea 7 (and determine if $\Pr[\; '] > p$) using an assignment that satisfies $'$ but does not satisfy $\quad$ as a certificate, assuming that such an assignment exists.

Lemma 4.6. For any fixed integer $k \quad 1$, kSAT-Prob$_{>1=2}$ is in NP.

Proof. Fix an integer $k \quad 1$. We describe a linear time *verifier* which takes as input a k-CNF $'$ and a certificate c, and has the following behavior:

- if Pr[$'$] > 1=2, then there exists a certificate c for which the verifier return YES;
- if Pr[$'$] 1=2, then for every certificate c the verifier returns NO.

This will then imply that kSAT-Prob$_{>1=2}$ is in NP.

Let $'$ be an arbitrary k-CNF, and let c be an input certificate.

By Lemma 3.21, we know that there exists a parameter family P of constants such that Algorithm 3 solves kSAT-Prob $_{1=2}$ given P. Run Algorithm 3 with input parameters P on the formula $'$. By Theorem 2.15, this takes linear time.

If the algorithm returns NO, then Pr[$'$] < 1=2, so we can return NO.

Otherwise, the algorithm returned YES. The only place the algorithm could have returned YES is in step 5 of Algorithm 3. In this case, the algorithm constructed a formula obtained from taking $'$ and asserting some clauses, computed the exact value of Pr[ ], and saw that Pr[ ] 1=2. Since is equivalent to $'$ and the conjunction of some clauses, any solution to is a satisfying assignment of $'$. So Pr[$'$] Pr[ ] in this case.

If Algorithm 3 computed that Pr[ ] > 1=2, then Pr[$'$] > 1=2 as well, so we return YES.

Otherwise, Algorithm 3 computed that Pr[ ] = 1=2. In this case, Pr[$'$] > 1=2 if and only if there exists an assignment x which satisfies $'$ but does not satisfy . At this point, we check if the certificate c is an assignment x which satisfies $'$, but does not satisfy . If so, we return YES. If not, we return NO. Performing this check for x takes linear time, since we just go through the clauses to make sure the assignment satisfies each of them.

This works because if Pr[$'$] > 1=2, such an assignment x exists, so some certificate does let us return YES, and if Pr[$'$] = 1=2 no such assignment can exist (and thus we return NO for every certificate) because Pr[$'$] = Pr[ ] so the set of satisfying assignments for $'$ and are the same.

All the steps above run in linear time, so kSAT-Prob$_{>1=2}$ is in NP as claimed.

Finally, we show that for the special case of $k = 3$, we do not need a certificate to verify YES instances of kSAT-Prob$_{>1=2}$, and instead can solve the problem directly in linear time. Intuitively, this is because we can implement Idea 7 by looking for a satisfying assignment for a 2-CNF, which takes linear time by Proposition 2.12.

Lemma 4.7. For any positive integer $k \quad 3$, kSAT-Prob$_{>1=2}$ can be solved in linear time.

Proof. We describe a linear time algorithm which takes as input a 3-CNF $'$ and determines if Pr[$'$] > 1=2.

Let $'$ be the input 3-CNF formula.

By Lemma 3.14, there exist a set of constant parameters such that Algorithm 2 solves 3SAT-Prob$_{>1=2}$ when given these parameters. Run Algorithm 2 with these parameters on $'$. By Theorem 3.9, this takes linear time. If the algorithm returns NO, then Pr[$'$] < 1=2, so we return NO.

Otherwise, the algorithm returns YES. The only place the algorithm to returns YES is step 5 of Algorithm 2. Let $i \quad 1$ denote the value of the variable i in Algorithm 2 at the time

the algorithm halted. In this case, by eq. (15) from the proof of Lemma 3.14, the algorithm found a formula equivalent to

$$' \wedge \left(\bigwedge_{j=1}^{i \quad 1} `_j \right)$$

where the $`_j$ are distinct literals, computed the value of Pr[ ], and found that Pr[ ] 1=2. Since is equivalent to a formula containing ' as a subformula, we necessarily have

$$\Pr[\,'] \quad \Pr[\ ] \quad 1=2:$$

If Pr[ ] > 1=2, then Pr['] > 1=2 as well, so in this case we return YES.
Otherwise, Pr[ ] = 1=2.
If $i = 1$, then is equivalent to ', so we have Pr['] = Pr[ ] = 1=2 and return NO.
So suppose i 2. We claim that in fact $i = 2$.
Indeed, if $i \not= 2$ we would have i 3, but in this case

$$\Pr[\ ] \quad \Pr[`_1 \wedge `_2] \quad 1=4$$

which contradicts the assumption that Pr[ ] = 1=2.
Since $i = 2$, is equivalent to ' ^ $`_1$. For convenience, write ` = $`_1$.

B Claim 4.8. The literal ` appears in every clause of '.

Proof. Suppose to the contrary that ' has a clause C not containing `. Then there exists an assignment which sets ` to be true, yet does not satisfy C. Since half of all assignments to the variables of ' set ` to be true, and every solution to $C \setminus `$ must set ` true, we have

$$\Pr[C \wedge `] < 1=2:$$

Then by Proposition 2.16 we have

$$\Pr[\ ] = \Pr[' \wedge `] \quad \Pr[C \wedge `] < 1=2:$$

This contradicts the assumption that Pr[ ] = 1=2.
So our initial assumption was false, and every clause of ' contains ` as claimed. □

Since any satisfying assignment of ' sets ` to be either true or false, we have

$$\Pr[\,'] = \Pr[' \wedge `] + \Pr[' \wedge :`]$$

Since is equivalent to ' ^ ` and we already computed Pr[ ] = 1=2, the above equation implies that

$$\Pr[\,'] = (1=2) + \Pr[' \wedge :`]:$$

So Pr['] > 1=2 if and only if (' ^ :`) is satisfiable.
By Claim 4.8, every clause of ' contains `. Let

$$'^0 = fC \setminus f`g \mid C \in '\,g$$

be the formula obtained by removing ` from every clause in ′.

Since ′ is a 3-CNF, ′′ must be a 2-CNF.

Since ` appears in every clause of ′, the 3-CNF ′ ∧ :` is equivalent to the 2-CNF ′′ ∧ :`. So Pr[′] > 1=2 if and only if Pr[′′ ∧ :`] > 0. By Proposition 2.12 we can determine if

$$\Pr[\phi'' \wedge \neg \ell] > 0$$

in linear time. If Pr[′′ ∧ :`] > 0 we return YES, otherwise we return NO.

This solves 3SAT-Prob$_{>1=2}$ in all cases, and proves the desired result.

Proof of Theorem 4.4. The result follows immediately by combining Lemmas 4.5 to 4.7.

4.2 Limited Long Clauses

The Majority-SAT problem is PP-complete in general, but becomes polynomial-time solvable when restricted to k-CNF formulas for constant k, by Theorem 2.15. What about the case when the input formulas are not necessarily k-CNFs, but are "almost k-CNFs," in the sense that all but a smaller number of clauses in the formula have width at most k? Can we solve Majority-SAT on such formulas in polynomial-time as well?

It turns out the answer is no we cannot, at least for k 3.

Theorem 4.9: Threshold Satisfaction for 3-CNFs with One Long Clause

For positive integers k, the Majority-SAT problem restricted to formulas of the form ∧ L, where is a k-CNF and L is an arbitrary clause, is

- polynomial-time solvable for k 2,
- NP-hard under Turing reductions for k = 3, and
- PP-complete for k 4.

Theorem 4.9 is interesting it shows that the tractability of kSAT-Prob $_{1=2}$ for fixed k is quite brittle. Majority-SAT is easy on k-CNFs for constant k, but adding just *one clause of unbounded width* to the input formula can make the problem hard again.

We prove Theorem 4.9 across a series of lemmas.

We begin first by showing the hardness results in Theorem 4.9.

To show the PP-hardness result, we reduce from the following PP-complete problem.

Proposition 4.10. Given a 3-CNF formula ′ on n variables and an integer a 2 [n], the problem of determining whether Pr[′] (1=2)^a is PP-complete.

The problem from the statement of Proposition 4.10 is in PP because it is a special case of Majority-SAT. The problem is PP-hard by [BDK07, Proposition 1], which shows that the problem is PP-hard even if we restrict to the special case where the input integer is of the form a = n(1 1=t), where n is the number of variables in the input formula ′ and t 2 [n].

Proposition 4.10 then holds because, by the discussion from the previous paragraph, the problem from its statement is PP-hard and in PP.

Note the difference between kSAT-Prob $_p$ and the problem from Proposition 4.10: the latter requires us to check if the satisfaction probability of a 3-CNF is at least $1=2^t$ where t is given as part of the input, and so can equal *any* positive integer. In particular, the threshold we test at in the problem can depend on n, and thus does not have to be a constant. For example, solving the problem for $t = n$ corresponds to solving 3SAT, which is NP-hard.

To prove hardness for solving Majority-SAT on k-CNF formulas adjoined to a single "long clause" L, we use the intuition from Idea 6 and show the following general reduction.

Lemma 4.11. For any integer k 2, there is an algorithm which given a $(k$ 1)-CNF $'$ and integer a 1, in $O(j\ 'j + a)$ time outputs a k-CNF and clause L such that

$$\Pr[\ '] \quad (1=2)^a \ (\ ) \ \Pr[\ \ ^\ L] \quad 1=2:$$

Proof. Let $'$ be an arbitrary $(k$ 1)-CNF. Let a 1 be an integer.

Let $x; y_1; : : : ; y_a$ be distinct variables not appearing in $'$. Construct the k-CNF formula

$$= fC \ [\ fxg \ j \ C \ 2 \ 'g$$

by adding x to each clause of $'$. Furthermore, define

$$L = (:x) \_ \left(\bigvee_{i=1}^{a} y_i \right)$$

to be a clause on $a + 1$ variables.

Given $'$ and a, we can construct and L in $O(j\ 'j + a)$ time.

Since every assignment sets x to be either true or false, we have

$$\Pr[\ \ ^\ L] = \Pr[\ \ ^\ L \ ^\ x] + \Pr[\ \ ^\ L \ ^\ :x]: \tag{47}$$

Since x appears in every clause of , we have

$$\Pr[\ \ ^\ L \ ^\ x] = \Pr[L \ ^\ x] = \Pr[(y_1 \_ \quad \_ \ y_a) \ ^\ x] = (1=2) \ \ (1 \quad (1=2)^a) : \tag{48}$$

Since L contains $:x$, we have

$$\Pr[\ \ ^\ L \ ^\ :x] = \Pr[\ \ ^\ :x] = \Pr[\ ' \ ^\ :x] = (1=2) \ \ \Pr[\ ']: \tag{49}$$

Combining eqs. (47) to (49) we get that

$$\Pr[\ \ ^\ L] = (1=2) \ \ (1 + (\Pr[\ '] \quad (1=2)^a)) :$$

This means that $\Pr[\ \ ^\ L]$ $1=2$ if and only if $\Pr[\ ']$ $(1=2)^a$, as desired.

Corollary 4.12. The Majority-SAT problem on formulas of the form $\ ^\ L$, where is a 4-CNF and L is an arbitrary clause, is PP-complete.

Proof. By Lemma 4.11, there is a polynomial-time reduction from the problem of deciding if 3-CNF $'$ on n variables has $\Pr[\ ']$ $(1=2)^a$ for some integer a 2 $[n]$ to the problem of deciding if $\Pr[\ \ ^\ L]$ $1=2$ for some 4-CNF and clause L.

By Proposition 4.10, the first problem from the previous paragraph is PP-hard. Thus the second problem from the previous paragraph must also be PP-hard. The second problem is a special case of Majority-SAT, and thus is in PP. So the second problem is PP-complete, which proves the desired result.

Lemma 4.13. The Majority-SAT problem on formulas of the form $\wedge L$, where is a 3-CNF and L is an arbitrary clause, is NP-hard under Turing reductions.

Proof. By [Zuc96, Theorem 4.1], the following problem is NP-hard: given a 2-CNF $'$, find a real $\hat{p}$ 2 $[0;1]$ such that $\hat{p}$ $\Pr[']$ $2\hat{p}$. In other word approximating the fraction of satisfying assignments of a 2-CNF to a factor of two is NP-hard. To prove the result, it suffices to present a Turing reduction from this problem to the problem in lemma statement.

Take an arbitrary 2-CNF formula $'$ on n variables. Fix an integer a 2 $[n]$. Then by Lemma 4.11, we can in polynomial time construct a 3-CNF $_a$ and a clause L_a with the property that $\Pr[']$ $(1=2)^a$ if and only if $\Pr[_a \wedge L_a]$ $1=2$. Using a subroutine which solves the problem from the lemma statement, we can determine for each a 2 $[n]$ whether

$$\Pr[_a \wedge L_a] \quad 1=2$$

holds.

If the above inequality does not hold for any a 2 $[n]$, then $\Pr[']$ $<$ $(1=2)^n$. Since $'$ has n variables, this means that $\Pr[']$ $=$ 0, so we can return $\hat{p}$ $=$ 0 in this case.

Otherwise, the above inequality holds for some a 2 $[n]$. Then we can find the largest positive integer b such that $\Pr[_b \wedge L_b]$ $1=2$. Then by the equivalence from the previous paragraph, we have

$$(1=2)^b \quad \Pr[\,'\,] < 2 \quad (1=2)^b$$

so we can take $\hat{p}$ $=$ $(1=2)^b$.

This completes the polynomial-time Turing reduction, and proves the desired result.

It remains to show how to solve Majority-SAT in polynomial time over CNF formulas where all but one clause has width at most two. We in fact show a more general result below, where we can test at thresholds p $\neq$ $1=2$, and allow the formula to have $O(\log n)$ long clauses.

Lemma 4.14. For any real p 2 $(0;1)$, and integer r 1, given a CNF formula of the form $\wedge$, where is a 2-CNF and has at most r clauses, we can determine if $\Pr[\wedge]$ p in 2^r $\mathrm{poly}(1=p)$ $(j \; j + j \; j)$ time.

Proof. We begin by running Algorithm 1 on . By Lemma 3.8 and the proof of Theorem 3.1, this takes $2^{O(\log(1=p))} j \; j$ time and determines if $\Pr[\,]$ p.

If Algorithm 1 returns NO, then $\Pr[\,]$ $<$ p, so

$$\Pr[\wedge] \quad \Pr[\,] < p$$

by Proposition 2.16.

Otherwise, Algorithm 1 returns YES. In this case, we must have found a set H of at most $O(\log(1=p))$ variables of in step 4 of Algorithm 1, with the useful property that for every assignment $: H$ $f0;1g$, the induced formula is a 1-CNF.

By Proposition 2.18, for any fixed $: H$ $f0;1g$, the satisfying assignments of $\wedge$ are precisely the satisfying assignments of $\wedge$ which agree with on H. Since every assignment to the variables of restricts to some unique assignment on H, we have

$$\Pr[\wedge] = \sum_{: H \to f0;1g} \Pr[\wedge]: \tag{50}$$

Now, let

$$= \bigwedge_{i=1}^{r} C_i:$$

By the principle of inclusion-exclusion applied to the event that each C_i is satisfied by an assignment, we have

$$\Pr[\quad \wedge \quad] = \sum_{S \quad [r]} (\quad 1)^{jSj} \Pr\left[\quad \wedge \left(\bigwedge_{i2S} (: C_i)\right)\right] \tag{51}$$

for any $\quad : H \quad f0; 1g$.

A clause

$$C = (`_1 \_ \quad \_ `_w)$$

is satisfied precisely when at least one of its literals $`_i$ is satisfied. This means that its negation, $: C$, is satisfied precisely when all literals $`_i$ are false.

So $: C$ is equivalent to the 1-CNF formula

$$\bigwedge_{i=1}^{w} (: `_i):$$

This means that for each $S \quad [r]$, the formula

$$\bigwedge_{i2S} (: C_i)$$

is equivalent to a 1-CNF. We also know that for each $\quad : H \quad f0; 1g$, the induced formula is a 1-CNF. So by Proposition 2.11, we can compute the satisfaction probability appearing in each summand in the right-hand side of eq. (51) in linear time.

Thus by eq. (51) we can compute

$$\Pr[\quad \wedge \quad]$$

for any fixed $\quad : H \quad f0; 1g$ in $2^r(j \quad j + j \quad j)$ time asymptotically. Doing this for all assignments $\quad$ to the variables in H then lets us compute $\Pr[\quad \wedge \quad]$ using eq. (50) in

$$2^{O(\log(1=p))} 2^r (j \quad j + j \quad j)$$

time. We can then return YES if $\Pr[\quad \wedge \quad] \quad p$, and return NO otherwise.

Since $2^{O(\log(1=p))} \quad \text{poly}(1=p)$, we get that the total runtime of the algorithm is at most

$$2^r \, \text{poly}(1=p) \, (j \quad j + j \quad j)$$

as claimed.

Proof of Theorem 4.9. The theorem follows by combining Corollary 4.12, Lemma 4.13, and Lemma 4.14 for the special case of $p = 1=2$ and $r = 1$.

4.3 Existential

An interesting generalization of Majority-SAT is the Existential Majority-SAT problem, where we are given a formula on two disjoint sets of variables x and y, and are tasked with determining if it is possible to set the values of the variables in y to obtain a formula on the remaining y variables, whose fraction of satisfying assignments is at least p.

> **Existential Majority-SAT**
>
> Given a CNF formula $'(x;y)$ on $n = n_1 + n_2$ variables
>
> $$x = (x_1;\ldots;x_{n_1}) \quad \text{and} \quad y = (y_1;\ldots;y_{n_2});$$
>
> determine if there exists $a \in \{0,1\}^{n_1}$ such that the formula $'(a;y)$ on n_2 variables has
>
> $$\Pr[\,'(a;y)] \quad 1=2:$$

Existential Majority-SAT is an important problem for showing hardness of tasks in planning and scheduling problems (see e.g., [PD04, Dar09]). Intuitively, Existential Majority-SAT is relevant in these contexts because the problem is an abstraction of the setting where an agent wants to make some choices (i.e., assign values to variables in x) to maximize the probability they are successful in some goal (i.e., satisfy $'(x;y)$) over the randomness of the environment (i.e., a random assignment of values to the variables in y).

Existential Majority-SAT is complete for the class $\mathsf{NP}^{\mathsf{PP}}$ [LGM98], which is intuitively the class of decision problems which can be solved by a deterministic polynomial-time verifier, which has oracle access to an algorithm solving Majority-SAT.

Analogous to how we went from Majority-SAT to kSAT-Prob$_p$, we can go from Existential Majority-SAT to kSAT-9Prob$_p$, by restricting the problem k-CNF formulas, and allowing thresholds $p \in (0;1)$ beyond just 1=2.

> **kSAT-9Prob$_p$**
>
> Given a real $p \in (0;1)$ and a k-CNF formula $'(x;y)$ on $n = n_1 + n_2$ variables
>
> $$x = (x_1;\ldots;x_{n_1}) \quad \text{and} \quad y = (y_1;\ldots;y_{n_2});$$
>
> determine if there exists $a \in \{0,1\}^{n_1}$ such that the formula $'(a;y)$ on n_2 variables has
>
> $$\Pr[\,'(a;y)] \quad p:$$

What is the complexity of kSAT-9Prob$_p$ for constant k and p? Well, since Majority-SAT is PP-complete, but kSAT-Prob$_p$ is in P for constant k and p, by analogy we might expect that Existential Majority-SAT, which is $\mathsf{NP}^{\mathsf{PP}}$-complete over general CNF formulas, might decrease in complexity down to NP or even P when relaxed to kSAT-9Prob$_p$ for constant k and p. This is indeed what happens.

Theorem 4.15: Existential Majority-SAT is Hard for 3-CNFs

For any fixed integer $k \geq 3$ and constant $p \in (0,1)$, kSAT-$\exists$Prob$_p$ is NP-complete.

Proof. The problem is in NP, because given a certificate $a \in \{0,1\}^{n_1}$, by Theorem 2.15 we can check in polynomial time whether $\Pr[\varphi'(a,y)] \geq p$ holds.

The problem is NP-hard by reduction from kSAT (which is NP-hard for $k \geq 3$).

Take an arbitrary k-CNF formula $\varphi'(x)$ on n_1 variables.

Set $p = 1/2$, $n_2 = 1$, and let y be a variable not in the variable set x.

Define the k-CNF formula $\varphi(x,y) = \varphi'(x)$ whose value does not depend on y.

If φ' is satisfiable, then there exists an assignment $a \in \{0,1\}^{n_1}$, such that $\varphi(a,y)$ is a tautology, and so has satisfaction probability 1. If φ' is not satisfiable, then for every assignment $a \in \{0,1\}^{n_1}$, the formula $\varphi(a,y)$ is unsatisfiable as well, and thus has satisfaction probability 0. So the answer to kSAT-$\exists$Prob$_p$ on φ is YES if and only if φ' is satisfiable (for any choice of $p \in (0,1)$). This gives a reduction from kSAT to kSAT-$\exists$Prob$_p$ and proves the desired result.

The hardness for kSAT-$\exists$Prob$_p$ in Theorem 4.15 for $k \geq 3$ is just coming from the fact that kSAT-$\exists$Prob$_p$ has existential quantification, and kSAT is NP-hard for $k \geq 3$.

Since 2SAT is in P, we might then expect that 2SAT-$\exists$Prob$_p$ for constant p can be solved in polynomial-time as well. This turns out to indeed be true. The high-level idea is to apply the 2SAT-Prob$_p$ algorithm over the y variables in concert with the the 2SAT algorithm over the x variables.

Theorem 4.16: Existential Majority-SAT is Easy for 2-CNFs

We can solve 2SAT-$\exists$Prob$_p$ in $n^{O(\log(1/p))}|\varphi'|$ time.

Proof. Let $\varphi'(x,y)$ be an arbitrary 2-CNF formula. For convenience, we refer to the x variables as *outer* variables, and the y as *inner* variables. To solve 2SAT-$\exists$Prob$_p$, we need to determine if there exists some assignment a to the outer variables such that the resulting formula $\varphi'(a,y)$ over the inner variables has satisfaction probability at least p.

We can write

$$\varphi'(x,y) = \varphi'_{\text{out}}(x) \wedge \varphi'_{\text{mix}}(x,y) \wedge \varphi'_{\text{in}}(y)$$

where $\varphi'_{\text{in}}(y)$ consists of all clauses in φ' which use only inner variables, $\varphi'_{\text{out}}(y)$ consists of all clauses in φ' which use only outer variables, and $\varphi'_{\text{mix}}(x,y)$ consists of the remaining clauses, which contain exactly one literal from x and one literal from y.

Run Algorithm 1 to solve 2SAT-Prob$_p$ on φ'_{in}. From the proof of Theorem 3.1, we can run this algorithm in $\text{poly}(1/p)|\varphi'|$ time.

If Algorithm 1 returns NO, then $\Pr[\varphi'_{\text{in}}] < p$, so by Proposition 2.16 we have

$$\Pr[\varphi'(a,y)] \geq \Pr[\varphi'_{\text{in}}(y)] < p$$

for all $a \in \{0,1\}^{n_1}$, so we can return NO in the 2SAT-$\exists$Prob$_p$ problem.

Otherwise, Algorithm 1 returns YES. In this case, step 4 of Algorithm 1 finds a set H_{in} of at most $O(\log(1/p))$ variables in y, such that for every assignment $\sigma : H_{\text{in}} \to \{0,1\}$, the

induced formula $(\ '_{\text{in}})$ is a 1-CNF. We can use H_{in} to help compute satisfaction probabilities of $\Pr[\ '(a;y)]$ for various assignments a, thanks to the following claim.

B Claim 4.17. For any assignment a to the outer variables which satisfies $'_{\text{out}}$, we have

$$\Pr[\ '(a;y)] = \sum_{\ : H_{\text{in}} \ \square\ f0;1g} \Pr[\ '_{\text{mix}}(a;y) \wedge (\ '_{\text{in}}(y))\ ]$$

Proof. If a satisfies $'_{\text{out}}$, then $'(a;y)$ is equivalent to

$$'_{\text{mix}}(y;a) \wedge\ '_{\text{in}}(y):$$

Now, by Proposition 2.18, we know that for any $\ : H_{\text{in}}\ \ f0;1g$, the satisfying assignments of $'_{\text{mix}}(a;y) \wedge (\ '_{\text{in}}(y))$ are precisely the satisfying assignments of $'_{\text{mix}}(a;y) \wedge (\ '_{\text{in}}(y))$ which agree with $\ $ on H_{in}. Since every assignment over the inner variables restricts to a unique assignment on H_{in} we have

$$\Pr[\ '_{\text{mix}}(a;y) \wedge (\ '_{\text{in}}(y))] = \sum_{\ : H_{\text{in}} \ \square\ f0;1g} \Pr[\ '_{\text{mix}}(a;y) \wedge (\ '_{\text{in}}(y))\ ]:$$

By the first paragraph of this proof, this shows the claim. □

Now, for any assignment $a\ 2\ f0;1g^{n_1}$ to the outer variables, define $L(a)$ to be the set of literals $`_y$ over $y$ such that $'_{\text{mix}}$ contains a clause of the form $(`_x \_\ `_y)$ for some literal $`_x$ that a sets false. Equivalently, $L(a)$ is the set of literals which appear in unit clauses of $'_{\text{mix}}(a;y)$. By Proposition 2.16 we have

$$\Pr[\ '(a;y)]\quad \Pr[\ '_{\text{mix}}(a;y)] < (1=2)^{jL(a)j}:$$

From the above equation, we deduce that

$$\Pr[(a;y)]\quad p\ =)\ \ jL(a)j\quad b\log(1=p)c: \tag{52}$$

From eq. (52), we see that to solve $2\text{SAT-}9\text{Prob}_{\ p}$ on $'$, we may restrict our attention to assignments a to the outer variables for which $L(a)$ at most $b\log(1=p)c$ literals. This suggest a general algorithm strategy: try out all possible sets L of at most $b\log(1=p)c$ literals, and for each search for an assignment a with $L = L(a)$ such that $\Pr[\ '(a;y)]\quad p$.

To help describe this procedure, we introduce some additional notation. Let L_{mix} denote the set of literals $`_y$ over $y$ which appear in clauses of $'_{\text{mix}}$. For each literal $`_y\ 2\ L_{\text{mix}}$, we let $X(`_y)$ denote the set of literals $`_x$ over x such that $(`_x \_\ `_y)$ is a clause in $'_{\text{mix}}$. More generally, given $L\quad L_{\text{mix}}$ we define

$$X(L) = \bigcup_{`_y 2 L} X(`_y)$$

to be the set of outer variables appearing in some clause of $'_{\text{mix}}$ with a literal in L.

To help find assignments a with $L(a) = L$ for a given L, we prove the following claim.

Algorithm 4. Existential Threshold Satisfaction Algorithm for 2-CNFs

Inputs: A 2-CNF $'(x;y)$, and real $p \in (0;1)$.
Returns: YES if there exists $a \in \{0;1\}^{n_1}$ with $\Pr['(a;y)] \geq p$, NO otherwise.

1. For each $L \subseteq L_{\text{mix}}$ with $|L| \leq \lfloor \log(1/p) \rfloor$:
2. For each map $f : L \to X(L)$ satisfying $f(\ell_y) \in X(\ell_y)$ for every $\ell_y \in L$:
3. Determine if there exists $a \in \{0;1\}^n$ which sets all literals in $X(L_{\text{mix}} \setminus L)$ to be true, sets $f(\ell_y)$ to be false for every $\ell_y \in L$, and satisfies $'_{\text{out}}$.
4. If we find a satisfying the conditions from step 3, then compute

$$\Pr['(a;y)] = \sum_{: H_{\text{in}} \to \{0;1\}} \Pr['_{\text{mix}}(a;y) \wedge ('_{\text{in}}(y))]$$

and return YES if this if the sum is at least p.

5. Return NO.

Claim 4.18. Let $L \subseteq L_{\text{mix}}$ be a set of literals over y. An assignment a to the outer variables has $L(a) = L$ if and only if

1. for every $\ell_y \in L$, a sets some literal in $X(\ell_y)$ to be false, and
2. a sets every literal in $X(L_{\text{mix}} \setminus L)$ to be true.

Proof. For any literal ℓ_y in L_{mix}, ℓ_y appears in a unit clause of $'_{\text{mix}}(a;y)$ if and only if a sets some literal in $X(\ell_y)$ to be false. Since $L(a)$ is defined to be the set of literals over y which appear in unit clauses of $'_{\text{mix}}(a;y)$, the desired result follows. □

We present the remainder of the algorithm for solving 2SAT-Prob$_p$ in Algorithm 4. We first prove that the algorithm is correct, and then analyze its runtime.

Proof of Correctness

Suppose there exists an assignment a to the outer variables such that $\Pr['(a;y)] \geq p$. Then by eq. (52) we must have $|L(a)| \leq \lfloor \log(1/p) \rfloor$. Moreover, $L(a) \subseteq L_{\text{mix}}$ by definition. So some iteration of the loop in step 1 of Algorithm 4 sets $L = L(a)$. Consider this iteration.

By Claim 4.18, a must set some literal in $X(\ell_y)$ to be false for each $\ell_y \in L$. Consider a choice of f in step 2 of Algorithm 4 for which $f(\ell_y)$ is set to false by a for each $\ell_y \in L$.

By Claim 4.18, a must set every literal in $X(L_{\text{mix}} \setminus L)$ to be true. Also, the assignment a must satisfy $'_{\text{out}}$, since if $'_{\text{out}}$ is not satisfied by a, $\Pr['(a;y)] = 0$. Thus, a satisfies all conditions in step 3 of Algorithm 4.

Then in step 4 of Algorithm 4, we correctly compute $\Pr['(a;y)]$ by Claim 4.17, and so return YES in this case.

Conversely, suppose Algorithm 4 returns YES. Then it must do so in step 4. This can only happen if we found an assignment a satisfying the conditions of step 3 of Algorithm 4. This means that a satisfies $'_{\text{out}}$, so by Claim 4.17, step 4 computes that $\Pr[\,'(a;y)] \quad p$, so $'$ is indeed a YES instance for the 2SAT-9Prob $_p$ problem.

Thus the algorithm correctly solves 2SAT-9Prob $_p$ as claimed.

Runtime Analysis

The initial step of running Algorithm 1 on $'_{\text{in}}$ took $\text{poly}(1=p)j\,'j$ time.

There are at most $(2n)^{b\log(1=p)c}$ choices of L in step 1 of Algorithm 4. For each such choice of L, there are at most $(2n)^{jLj} \quad (2n)^{b\log(1=p)c}$ choices of function f in step 2 of Algorithm 4.

Fix a choice of L and f from the first two steps of Algorithm 4.

Step 3 of Algorithm 4 can be implemented in linear time. We first assign each literal in $X(L_{\text{mix}} \; n \; L)$ to be true and assign $f(`_y)$ to be false for each $`_y \; 2 \; L$. If this leads to an inconsistency (i.e., requires some variable to be set to both true and false simultaneously), then no such assignment exists. Otherwise, we get a partial assignment to a subset of outer variables from the procedure so far. We then construct the induced 2-CNF $(\,'_{\text{out}})$, and check if it is satisfiable in linear time by Proposition 2.12. This formula is satisfiable if and only if some assignment extending satisfies $'_{\text{out}}$, so an assignment a satisfying the conditions of step 3 from Algorithm 4 exists if and only if this procedure finds such an assignment.

By definition of $'_{\text{mix}}$, the formula $'_{\text{mix}}(a;y)$ is a 1-CNF.

By the definition of H_{in}, the formula $(\,'_{\text{in}}(y))$ is a 1-CNF for each $: H_{\text{in}} \quad f0;1g$.

Thus, we can compute the value of each summand in the right-hand of the equation in step 4 of Algorithm 4 in linear time by Proposition 2.11. So step 4 of Algorithm 4 takes $2^{jH_{\text{in}}j}\,'j \quad \text{poly}(1=p)j\,'j$ time.

Step 5 of Algorithm 4 takes $O(1)$ time.

So overall, Algorithm 4 takes at most

$$(2n)^{b\log(1=p)c} \quad (2n)^{b\log(1=p)c} \quad \text{poly}(1=p)j\,'j \quad n^{O(\log(1=p))}j\,'j$$

time as claimed.

4.4 Inference

In the Bayesian Inference problem, we are given CNF formulas $'$ and over a common set of variables and a real $p \; 2 \; (0;1)$, and are tasked with determining if

$$\Pr[\,' \wedge \,] \quad p \quad \Pr[\,]:$$

In the special case where is satisfied by every assignment (for example, if is the empty formula) and $p = 1=2$, Bayesian Inference recovers the Majority-SAT problem, and so in general is PP-hard. The name of this problem comes from the fact that if $\Pr[\,] \not= 0$, the above inequality is equivalent to the condition that

$$\Pr[\,' \; j \,] \quad p$$

and so the problem is intuitively asking us to infer how likely a uniform random assignment is to satisfy $'$, given that we know the assignment satisfies .

In light of our algorithms for kSAT-Prob $_p$ for constant k and p, it is natural to ask if the Bayesian Inference problem can be solved in polynomial time when p is a constant, and the formulas $'$ and have constant width.

We show that even if we limit $'$ to be a *single unit clause*, the Bayesian Inference problem becomes PP-hard for 3-CNFs . Previously, it was only known that this problem is NP-hard under Turing reductions [AW22, Section V].

The idea of the proof is to use a 3-CNF to check correctness of the tableau for an arbitrary circuit, and $'$ check that the circuit is satisfied (i.e., returns 1 at its output gate). The proof is due to Olaf Beyersdorff, Till Tantau, and Quinten Tupker, from a discussion at Dagstuhl Seminar 23111.

Theorem 4.19: Bayesian Inference for 1-CNFs Conditioned on 3-CNFs is Hard

The Bayesian Inference problem with threshold $p = 1/2$ restricted to inputs where $'$ is 1-CNF with one clause and is a 3-CNF is PP-hard.

To prove Theorem 4.19, we will reduce from a variant of Majority-SAT, defined over general circuits instead of CNF formulas.

Definition 4.20 (Boolean Circuit). A Boolean circuit C over variables $\vec{x} = (x_1, \ldots, x_n)$ of size s n is defined by a sequence of *gates* $G_1, \ldots, G_s$ with the following data:

- for $i \in [n]$, we have $G_i = x_i$ (the variable gates),
- for i $n+1$, each G_i is labeled as an *and* ($\wedge$) gate, *or* ($\vee$) gate, or *not* ($\neg$) gate,
- each and gate G_i, comes with indices $j, l < i$ such that $G_i = G_j \wedge G_l$,
- each or gate G_i, comes with indices $j, l < i$ such that $G_i = G_j \vee G_l$,
- each not gate G_i, comes with an index $j < i$ such that $G_i = \neg G_j$, and
- G_s is additionally labeled as the *output* gate.

Each gate G_i naturally computes a function over the variables $\vec{x}$, by setting $G_i = x_i$ for $i \in [n]$ to be the function that outputs the i^{th} coordinate of the input, and inductively defining the functions G_i for $i > n$ based off the ($\wedge$), ($\vee$), and ($\neg$) cases in terms of the functions computed by the G_j gates $j < i$ in the natural way. We say the function computed by the output gate G_s is the function computed by the circuit C.

Majority-Circuit SAT

Given a Boolean circuit C of size s on n variables, determine if $\Pr[C]$ $1/2$.

Proposition 4.21. Majority-Circuit SAT is PP-complete.

Proof. We first show that Majority-Circuit SAT is in PP. Consider the verifier which, given the input circuit C, returns YES if and only if the certificate is a satisfying assignment $a \in \{0,1\}^n$ of C. The verifier runs in polynomial time, because we can compute $C(a)$ by inductively computing the value $G_i(a)$ at each gate on the assignment, for $i \in [s]$ in increasing order. Computing $G_i(a)$ for each $i \in [n]$ takes $O(1)$ time since we just need to read the i^{th} coordinate of a. Computing $G_i(a)$ for $i > n$ takes $O(1)$ time, since we just to compute a logical operation on the values of $G_j(a)$ for at most two indices $j < i$, which we will have already computed by the time we reach gate G_i. So we can compute $G_s(a) = C(a)$, in polynomial time. By construction, at least half of the certificates in $\{0,1\}^n$ make the verifier return YES if and only if $\Pr[C]$ 1=2, so Majority-Circuit SAT is in PP as claimed.

It remains to show that Majority-Circuit SAT is PP-hard. We prove this by reduction from the Majority-SAT problem.

To reduce Majority-SAT to Majority-Circuit SAT, we use simple binary tree-based constructions which let us model CNF formulas as circuits of fan in-two.

B Claim 4.22. Given an integer w 1, there is an $O(w)$ time algorithm which constructs a Boolean circuit C of size $O(w)$ computing the conjunction of w given inputs.

Proof. If $w = 1$, we return the circuit of size one which returns the input.

If $w = 2$, we return the circuit of size three with $G_3 = G_2 \wedge G_1$.

If instead w 3, partition the input variables into sets X and Y of size $|X| = \lfloor w/2 \rfloor$ and $|Y| = \lceil w/2 \rceil$, recursively compute circuits C_X and C_Y computing the conjunctions of the variables in X and Y, and then return the circuit C which computes $C_X \wedge C_Y$, by ordering all non-variable gates of C_X before all non-variable gates of C_Y, and adding a new final gate G which computes the logical and of the output gates from C_X and C_Y.

An easy induction argument shows that this procedure runs in $O(w)$ time and outputs a circuit C with the desired property of size $O(w)$. □

B Claim 4.23. Given an integer w 1, there is an $O(w)$ time algorithm which constructs a Boolean circuit C of size $O(w)$ computing the disjunction of w given inputs.

Proof. Follows from symmetric reasoning to the proof of Claim 4.22. □

Take an arbitrary instance of Majority-SAT, consisting of a CNF $'$ over n variables.

We construct a Boolean circuit C equivalent to $'$, as follows.

Let $x_1, \ldots, x_n$ be the variables of $'$. The circuit C has this same variable set, and so we set $G_i = x_i$ for $i \in [n]$ as its initial variable gates. For each $i \in [n]$, we set $G_{n+i} = \neg G_i$ to be a not gate computing the literal $\neg x_i$.

Now, for each clause B in $'$, we use Claim 4.23 to construct a circuit G_B equivalent to B (the inputs to G_B are the variable and literal gates we already computed for C, corresponding to the literals appearing in B). We order these G_B gates arbitrarily after all the variable and literal gates of C. We then use Claim 4.22 to compute a circuit C^0 which computes the conjunction of the output gates of all the G_B circuits. We order the gates of C^0 after the gates of all the G_B circuits.

This completes the description of circuit C. The size and runtime bounds from Claims 4.22 and 4.23 show that we can compute C in polynomial time, and C has size at most polynomial in n and $|'|$. Circuit C is equivalent to $'$, because an assignment satisfies C if and only if

the output gate of every G_B circuit returns 1, which is equivalent to every clause of $'$ being satisfied. Since C and $'$ have the same number of variables, we then have $\Pr[C]$ $1=2$ if and only if $\Pr[']$ $1=2$, which proves the desired result.

Proof of Theorem 4.19. We will prove the result by reduction from Majority-Circuit SAT. By Proposition 4.21, Majority-Circuit SAT is PP-hard, so this will show the desired result.

Take an arbitrary instance of Majority-Circuit SAT, consisting of a Boolean circuit C of size s on n variables. Let $G_1; : : : ; G_s$ be the gates of C, ordered as in Definition 4.20. For each i 2 [s], introduce a variable g_i. We construct CNF formulas $'$ and over the g_i variables.

The formula $'$ is a 1-CNF consisting only of the unit clause $fg_s g$.

Next, we describe how to construct the 3-CNF formula .

Given any variables $x; y; z$, we define the 3-CNF formula

$$_{\wedge}(x; y; z) = (x \_ : y \_ : z) \wedge (: x \_ y) \wedge (: x \_ z) :$$

By inspection, $_{\wedge}(x; y; z)$ is equivalent to the condition that $[\![x = y \wedge z]\!]$ (i.e., an assignment satisfies $_{\wedge}(x; y; z)$ precisely when x is assigned the conjunction of the values assigned to y and z). We similarly define the 3-CNF formula

$$_{\_}(x; y; z) = (: x \_ y \_ z) \wedge (x \_ : y) \wedge (x \_ : z)$$

which, by inspection, is equivalent to the condition that $[\![x = y \_ z]\!]$.

Given variables x and y, we also define the 2-CNF formula

$$_{:}(x; y) = (x \_ : y) \wedge (: x \_ y) :$$

By inspection, $_{:}(x; y)$ is equivalent to the condition that $[\![x = : y]\!]$.

Now, for each i 2 [s] with $i > n$,

- if G_i is an and gate with $G_i = G_j \wedge G_l$, we define the formula

$$_i = \ _{\wedge}(g_i; g_j; g_k);$$

- if G_i is an or gate with $G_i = G_j \_ G_l$, we define the formula

$$_i = \ _{\_}(g_i; g_j; g_k);$$

- and if instead G_i is a not gate with $G_i = : G_j$, we define the formula

$$_i = \ _{:}(g_i; g_j) :$$

We then define

$$= \bigwedge_{i=n+1}^{s} \ _i :$$

Given C, we can construct the 3-CNF in polynomial time.

We claim that an assignment a (where variable g_i is assigned value a_i) satisfies if and only if on input $(a_1; : : : ; a_n)$, gate G_i of circuit C computes the value a_i for all i 2 [s]. This

equivalence follows immediately from the construction of and an easy induction argument over the gates of circuit C.

Then, any assignment $(a_1; \ldots; a_n)$ to the variables g_i for $i \ 2 \ [n]$ extends to a unique satisfying assignment of . Consequently, we have

$$\Pr[\ ] = (1=2)^{s \ n}: \tag{53}$$

Recall that G_s is the output gate of C. By the above discussion, an assignment

$$a = (a_1; \ldots; a_s)$$

satisfies $' \wedge$ if and only if on input $(a_1; \ldots; a_n)$, gate G_i computes a_i for every $i \ 2 \ [s]$, and the circuit C outputs 1. The number of such assignments is just the number of satisfying assignments of C. Since C is a function over n variables, while $' \wedge$ is a formula over s variables, we have

$$\Pr[' \wedge \ ] = (2^n \ \Pr[C]) = (2^s) = (1=2)^{s \ n} \ \Pr[C]: \tag{54}$$

By comparing eqs. (53) and (54), we see that

$$\Pr[' \wedge \ ] \quad (1=2) \ \Pr[\ ]$$

if and only if

$$\Pr[C] \quad 1=2:$$

So we have a correct polynomial-time reduction from Majority-Circuit SAT to the problem from the statement of Theorem 4.19, which proves the desired result.

Chapter 5

Open Problems

Exact Parameterized Complexity

Although our algorithm for kSAT-Prob$_p$ runs in polynomial time for constant k and p, as discussed in the Exact Parameterized Complexity subsection of Section 3.4 this runtime grows rapidly as a function of p for $k \geq 3$. We can however solve 2SAT-Prob$_p$ on 2-CNFs $'$ in poly$(1/p)|'|$ time, which has reasonable dependence on p. Is a similar dependence on p possible for the general kSAT-Prob$_p$ problem, or can we prove that a such a dependence is unlikely to exist?

Open Problem 1. Can kSAT-Prob$_p$ be solved in poly$(1/p)|'|$ for some fixed $k \geq 3$? What about achieving a $2^{\text{poly}(1/p)}|'|$ runtime?

One potential strategy to attack Open Problem 1 is to apply more effective versions of the sunflower lemma. The reasoning used in the proof of correctness for our algorithms uses techniques from the proof of the classic sunflower lemma of [ER60], and the effective bounds from this lemma are reflected in the parameterized time complexity of our kSAT-Prob$_p$ algorithms. In recent years, better bounds have been proven for the sunflower lemma (for example, see the discussion and references in [Rao22]). Can these more effective results help design faster parameterized algorithms for threshold satisfaction?

Open Problem 2. Can the ideas from improved sunflower theorems be used to design algorithms for kSAT-Prob$_p$, whose runtimes have better dependence on k and p?

In Section 4.3, we discussed Existential Majority-SAT, a generalization of Majority-SAT that involves existential quantification over certain variables in the input formula. We showed in Theorem 4.16 that 2SAT-9Prob$_p$, the variant of Existential Majority-SAT restricted to 2-CNFs that tests at a threshold of p, can be solved for formulas $'$ in $n^{O(\log(1/p))}|'|$ time. For any constant p this runtime is poly(n), but this polynomial runtime becomes slower as p becomes smaller. Is there an algorithm solving 2SAT-9Prob$_p$ faster in terms of p, which has a *fixed* polynomial runtime for all constant p?

Open Problem 3. Can 2SAT-9Prob $_p$ be solved in $f(1=p)j'j$ time for some computable function f? Or is this impossible under a plausible hardness hypothesis?

Spectral Gaps

In Idea 1 (from the end of Section 2.2), we mentioned that the possible values of k-CNF satisfaction probabilities are constrained compared to the probabilities achieved by general CNF formulas. Using the arguments from the proof of Theorem 2.15, one can show a precise sense in which k-CNF fractions of satisfying assignments are limited—below every p 2 $(0;1)$, there is a *gap*, whose size depends only on p, where no satisfaction probability of any k-CNF is present. This was proven formally in [Tan22b, Corollary 1.3]:

Proposition 5.1 (Spectral Gaps). For each positive integer k and real p 2 $(0;1)$, there exists a real = $\text{gap}_k(p) > 0$ such that no k-CNF ' has the property that Pr['] 2 $(p$;$p)$.

The constant $\text{gap}_k(p)$ is called the *spectral gap* for k-CNFs at p. The parameterized runtimes of the current fastest algorithms for kSAT-Prob $_p$ seem to be proportional to the value of $\text{gap}_k(p)$. This suggests that understanding the value of $\text{gap}_k(p)$, by obtaining better upper and lower bounds for it, in terms of k and p, is an interesting research question. For example, [Tan22b, Theorem 2.15] lower bounds $\text{gap}_k(p)$ in terms of some inverse exponential tower that depends on the binary expansion of p, and seems qualitatitively similar to the bounds derived in Section 3.4.

Open Problem 4. Can we get precise upper and lower bounds on the size of $\text{gap}_k(p)$ in terms of k and p?

More generally, understanding the behavior of $\text{gap}_k(p)$ and the possible fractions of satisfying assignments that can be achieved by k-CNF formulas for small constant k seems like an interesting research project, from a purely mathematical perspective.

Open Problem 5. For small constants k 2, are there simple descriptions for the set

$$f\text{Pr}[\,']\,j\ '\text{ is a } k\text{-CNF}g$$

of possible satisfaction probabilities for k-CNF formulas?

Nested Majorities

In Section 4.3, we discussed the Existential Majority-SAT problem, a "higher-order" variant of Majority-SAT which introduces existentially quantified variables. Maj-Maj-SAT is another higher-order variant of Majority-SAT, relevant in the literature of probabilistic planning and inference (see e.g., [PD04, Dar21]). This problem is defined over CNFs over disjoint sets of variables x and y, and asks if at least half of the assignments to the x variables produce formulas over y with satisfaction probability at least 1=2.

Maj-Maj-SAT

Given a CNF formula $\varphi(x, y)$ on $n = n_1 + n_2$ variables

$$x = (x_1, \ldots, x_{n_1}) \quad \text{and} \quad y = (y_1, \ldots, y_{n_2}),$$

determine if

$$\Pr_a \left[\Pr_b [\varphi(a, b)] \geq 1/2 \right] \geq 1/2.$$

Just as Majority-SAT is PP-complete, Maj-Maj-SAT is complete for the class $\mathsf{PP}^{\mathsf{PP}}$ [Wag86, Theorem 7], a complexity class believed to contain problems much harder than those in PP. Intuitively, $\mathsf{PP}^{\mathsf{PP}}$ is the class obtained by taking the definition of PP from Section 2.1 (between the definitions of #SAT and Majority-SAT), and granting the polynomial-time verifier in that definition query access to an oracle that solves Majority-SAT in constant time.

For any integer $k \geq 1$ and reals $p, q \in (0, 1)$, we can also consider the kSAT-Prob$_{p, q}$ problem, obtained by restricting Maj-Maj-SAT to k-CNFs, and replacing 1/2 with thresholds values p and q.

kSAT-Prob$_{p, q}$

Given a k-CNF formula $\varphi(x, y)$ on $n = n_1 + n_2$ variables

$$x = (x_1, \ldots, x_n) \quad \text{and} \quad y = (y_1, \ldots, y_n),$$

determine if

$$\Pr_a \left[\Pr_b [\varphi(a, b)] \geq q \right] \geq p.$$

Since kSAT-Prob$_p$ is polynomial-time solvable for constant k and p, it is natural to suspect that kSAT-Prob$_{p, q}$ for constants k, p, and q. This indeed turns out to be true, and [Tan22b, Theorem 3.15] gives an efficient reduction from kSAT-Prob$_{p, q}$ to lSAT-Prob$_p$, where l is a positive integer depending only on k and q. By inspecting the proofs in [Tan22b, Section 3.1], we find that the current reduction sets l to be the smallest positive integer with

$$l \geq \left(2 + 2 \log_{2^k/(2^k - 1)} (1/\mathrm{gap}_k(q))\right)^k k^k k! \tag{55}$$

where $\mathrm{gap}_k(q)$ is the spectral gap for k-CNFs at q.

Although for constant k and q the parameter l is still constant, in general l can be very large in terms of k and q, because $\mathrm{gap}_k(q)$ can be very small in terms of k and q. In order to design algorithms for kSAT-Prob$_{p, q}$ with better dependence on k, p, q, it would be interesting to see if the reduction to lSAT-Prob$_p$ could be improved, by decreasing the blow-up in parameter size from k to l.

Open Problem 6. Can we efficiently reduce kSAT-Prob$_{p, q}$ to lSAT-Prob$_p$ for some positive integer l that is significantly smaller than the right hand side of eq. (55)?

Since 2SAT-Prob $_p$ can be solved in poly$(1=p)j\ 'j$ time on 2-CNFs $'$, it would be interesting to see if 2SAT-Prob $_{p;\ q}$ can be solved similarly quickly, with only polynomial dependence on $(1=p)$ and $(1=q)$ in the runtime. The current fastest algorithm for 2SAT-Prob $_{p;\ q}$ reduces to lSAT-Prob $_p$ where $l\quad 3$ is a growing function of q, and so has terrible dependence on p and q, as suggested by the discussion in Section 3.4.

Open Problem 7. Can we design algorithms for 2SAT-Prob $_{p;\ q}$ with better runtime dependence on p and q? For example, can the 2SAT-Prob $_{p;\ q}$ problem be solved in poly$(1=p; 1=q)j\ 'j$ time on 2-CNFs $'$? Or is this runtime ruled out under some plausible hardness hypothesis?

Complexity Classification

In Lemma 4.13, we showed that solving Majority-SAT on CNF formulas where all but possibly one clause has width at most three is NP-hard (under Turing reductions). However, we did not place this problem in NP. So, it is conceivable that this problem could even be PP-hard.

Open Problem 8. Is Majority-SAT restricted to formulas of the form $\ ^\wedge L$, where $\ $ is a 3-CNF and L is an arbitrary clause, NP-complete? Or is this problem PP-complete?

In the proof of Lemma 4.13, we showed NP-hardness for the above problem by reducing from the task of approximating the fraction of satisfying assignments of a 2-CNF to a factor of two. A potential strategy to strengthen this result and show PP-hardness for the above problem (and thus resolve Open Problem 8) is to show that approximately counting satisfying assignments of 2-CNFs to a factor of two is not just NP-hard, but in fact PP-hard. This motivates the following problem.

Open Problem 9. Given a 2-CNF formula $'$ and an integer a, is the problem of deciding whether Pr$[\ ']\quad (1=2)^a$ in NP? Or is this problem PP-complete?

See also Proposition 4.10 for additional context on the task described in Open Problem 9.

In Lemma 4.14, we showed that given a CNF $'$ with at most r clauses of width greater than two, we can determine if Pr$[\ ']\quad p$ in 2^r poly$(1=p)j\ 'j$ time. In particular, this problem can be solved in polynomial time provided $r\quad O(\log n)$. Is a polynomial time algorithm also possible when r is super-logarithmic in n, or does this problem become hard?

Open Problem 10. Given constant $p\ 2\ (0; 1)$, for what values of r is the problem of determining if Pr$[\ ']\quad p$, for CNF formulas $'$ where all but r clauses have width at most two, in P? Can this problem be solved in polynomial time for some $r\quad !\ (\log n)$?

In Section 4.4 we showed that the Bayesian Inference problem is already PP-hard over input formulas $'$ and $\ $ when $'$ is a 1-CNF and $\ $ is a 3-CNF. If $'$ and $\ $ are both 1-CNFs, then Bayesian Inference can be solved in linear time, since we can compute the satisfaction probabilities Pr$[\ '\ ^\wedge\ \ ]$ and Pr$[\ \ ]$ in linear time by Proposition 2.11. It remains unclear however, if we can efficiently solve Bayesian Inference when $\ $ is a 2-CNF.

Open Problem 11. Can Bayesian Inference be solved in polynomial time when the input formulas are 2-CNFs and the threshold $p \in (0, 1)$ is constant? What about when $'$ is a 1-CNF but $$ can be a 2-CNF?

In general, showing hardness results for threshold problems on 2-CNF formulas seems difficult. This is because most hardness results involving counting satisfying assignments on 2-CNFs involve polynomial interpolation arguments, or other reductions which greatly distort the counts of satisfying assignments in strange ways [Val79a]. This disortion is necessary to some extent, since 2SAT is polynomial-time solvable.

Lower Bounds

In Open Problem 1, we asked whether the time complexity for solving kSAT-Prob$_p$ can be improved for fixed k, in terms of the parameter p. In the case that significantly faster algorithms are not possible for this problem, it would be interesting to develop a theory of lower bounds for threshold satisfaction problems. For example, there are many influential hardness hypotheses concerning the intractability of problems where one seeks to approximate the maximum number of constraints that can be satisfied in a CNF formula, such as the Gap Exponential Time Hypothesis (see e.g., [Din16, Hypothesis 2.5]) and the Parameterized Intractability Hypothesis (see e.g., [GLR+23, Hypothesis 1.1]). Could one of these hardness hypotheses be useful for showing hardness for kSAT-Prob$_p$? Can we define new, plausible hypotheses which imply conditional lower bounds for the time complexity of kSAT-Prob$_p$?

Open Problem 12. Is there a plausible hardness hypothesis we can use to lower bound the parameterized time complexity of kSAT-Prob$_p$ and its variants?

Open Problem 13. Can we prove that (assuming some plausible hardness hypothesis) the runtime of any algorithm solving kSAT-Prob$_p$ must depend on $\text{gap}_k(p)$ in some way?

It is known that kSAT-Prob$_p$ admits efficient *kernelization* (also known as compression) algorithms in the parameters k and p [Tan22a]. That is, for constant k and p, there is a polynomial-time algorithm which takes as input a k-CNF formula $'$ and produces a k-CNF formula $$ of *constant size*, such that $\Pr[']$ p if and only if $\Pr[\ ]$ p. However, the size $$ depends on the value of $\text{gap}_k(p)$, and so is quite large in terms of k and p. So it would be interesting to obtain more efficient compression for kSAT-Prob$_p$.

Open Problem 14. Can we design better kernelization algorithms for kSAT-Prob$_p$, which give better compression in terms of k and p?

Alternatively, it would also be very interesting to show that significantly better compression for kSAT-Prob$_p$ is impossible. Establishing lower bounds on the possibility of

compressing kSAT-Prob$_p$ instances in this case may be easier than trying to directly resolve Open Problems 12 and 13, since there already exists a well-developed toolkit for proving kernelization lower bounds of parameterized problems [FLSZ18, Part III].

Open Problem 15. Can we prove interesting kernelization lower bounds for the kSAT-Prob$_p$ problem in terms of the parameters k and p, under some popular hypothesis in complexity theory?

Regularity

In the Regularity subsection of Section 3.4, we proved Theorem 3.28, a "regularity lemma" for k-CNFs. It would be interesting to see if this regularity lemma could be applied to design faster algorithms for other problems involving k-CNFs.

Open Problem 16. Are there interesting applications of Theorem 3.28 for problems on k-CNFs besides kSAT-Prob$_p$?

Part II

Graphs

Chapter 6

Algebraic Framework

In Chapters 7 and 8, we present algorithms for problems related to detecting disjoint paths in graphs. The approaches we use in these chapters fall into the general framework of *algebraic graph algorithms*, a growing area of modern algorithm design that combines linear algebraic techniques with arguments from classical and algebraic combinatorics to solve graph theoretic problems faster.

This paradigm often applies to computational problems where we are tasked with detecting a special type of pattern in a graph. Rather than identify this pattern by using some search routine, or by building up the structure of the pattern iteratively, in an algebraic algorithm we solve the problem by considering a related *polynomial*, whose monomials enumerate the instances of the pattern we are trying to detect. In particular, this enumerating polynomial is designed to be nonzero if and only if an instance of the desired pattern exists in the input graph. We can check if a polynomial is nonzero by randomly evaluating it at a point, so if our enumerating polynomial can be evaluated efficiently, this gives a randomized algorithm for the original problem.

Why is working with an auxiliary polynomial useful? The main reason is that working with polynomials enables us to employ arguments related to *counting* patterns, and this opens up a larger space of algorithmic techniques than one would have access to if they restricted themselves to direct detection-based argument. For example, in Chapter 7 we use enumerative properties of determinants to find maximum flow values in unweighted graphs, and in Chapter 8 we enumerate disjoint pairs of paths by enumerating the complementary set of intersecting pairs of paths, which turns out to be easier to work with. These are examples of methods we can only employ because we are working with enumerating polynomials.

A more subtle reason that the algebraic method is useful is that the structure of certain computational problems seems to *simplify when reduced modulo two*. For example, computing the permanent of a matrix with $\{0, 1\}$ entries is #P-complete [Val79b], but computing the value of the permanent of a matrix with $\{0, 1\}$ entries modulo two takes polynomial time, since the permanent is the same as the determinant of a matrix modulo two, and efficient algorithms exist for computing determinants. In Chapter 7, we use the fact that enumerating polynomials for intersecting pairs of paths simplify modulo two to obtain our algorithms.

What's this chapter useful for?

This chapter covers preliminaries used in the arguments of Chapters 7 and 8.

If you want to read Chapter 7, you should read this chapter first.

If you just want to read Chapter 8, you should first read Section 6.1, and the Node-Based subsection of Section 6.2, but can skip Section 6.3.

Organization

In Section 6.1 we recall the definitions of walks and paths in graphs, and introduce some notation and facts for working with polynomials and matrices over fields. In Section 6.2 we describe the general framework for encoding families of walks in graphs using polynomials. In Section 6.3 we discuss standard results for formal power series, which are used in Chapter 7.

6.1 Preliminaries

Graph Notation and Terminology

Throughout this chapter, as well as Chapters 7 and 8, we work with an input graph G on n vertices and m edges. We let V and E denote the vertex and edge sets of G respectively.

Given an edge $e = (u; v) \in E$, we let $\mathsf{tail}(e) = u$ denote the node e exits, and $\mathsf{head}(e) = v$ denote the node e enters.

A *walk* in G is a sequence of vertices $W = \langle v_0, \ldots, v_r \rangle$ such that $(v_i; v_{i+1}) \in E$ for each nonnegative integer $i \leq r - 1$. If $v_0 = s$ and $v_r = t$, we say that walk W *begins* at s and *ends* at t. We also write that W is an $s \rightsquigarrow t$ walk. We can also naturally view W as a sequence of edges $W = \langle e_1, \ldots, e_r \rangle$ where $e_i = (v_{i-1}; v_i)$ for each $i \in [r]$. We say that W begins at e_1 and ends at e_r. The *length* r of a walk is the number of edges it contains. The *size* $(r + 1)$ of a walk is the number of vertices it contains.

A *path* is a walk whose vertices are all distinct.

A path P is an $s \rightsquigarrow t$ *shortest* path if it is a path of minimum length in G from s to t.

Given walks P and Q such that the final vertex of P and the first vertex of Q are the same, we define $P \circ Q$ to be the path obtained by concatenating P and Q.

Finite Field Computation

Throughout, we work over a finite field $\mathbb{F} = \mathbb{F}_{2^q}$ of characteristic two, where q is a positive integer whose value we set separately for each algorithm. In general, we will pick q large enough so that our algorithms are correct with high probability. In every case however, we will have $q \geq 24 \log(m + n)$.

We work in the Word-RAM model with words of size $O(\log(m+n))$. This is the standard model of computation, since $m+n$ is the input size of the graph G. In this model, after some initial sublinear amount of time spent preprocessing the field $\mathbb{F}$, addition and multiplication of elements in $\mathbb{F}$ take $O(1)$ time, and division by nonzero elements in $\mathbb{F}$ takes $O(1)$ time. This follows from [JX24, Lemma 2.6], since we have field size $|\mathbb{F}| = 2^q \geq (m + n)^{24}$.

Identity Testing

Our algorithms will rely on the fact that random evaluations of a low degree polynomial over a large field are nonzero with high probability.

Proposition 6.1. Let P be a nonzero r-variate polynomial of degree at most d. Then a uniform random evaluation of P over F^r is nonzero with probability at least $1 \quad d/|\mathsf{F}|$.

See [MR95, Theorem 7.2] for a proof of Proposition 6.1.

Matrix Notation

Given a square matrix $\mathbf{M}$, we let $\mathbf{M}^{-1}$ denote the inverse of $\mathbf{M}$ if it exists. We let $\mathbf{I}$ denote the identity matrix, whose dimensions will be clear from context.

We let ω denote the *exponent of matrix multiplication*, the smallest positive real such that two $n \quad n$ matrices over F can be multiplied in $n^{\omega+o(1)}$ time. For convenience, we write the time complexity of matrix-multiplication as $O(n^{\omega})$ instead of $n^{\omega+o(1)}$. We have $\omega \quad 2$, since just reading the entries of an $n \quad n$ matrix takes (n^2) time. The current fastest algorithm for matrix multiplication implies that $\omega < 2.371552$ [WXXZ24].

6.2 Enumerating Families of Walks

We enumerate walks not by counting their number, but by assigning each walk a certain monomial weight, which records information about the vertices or edges traversed in the walk. Enumeration corresponds to summing the weights of all walks (or collections of walks) in a family of interest.

Node-Based

For every edge $(u,v) \in E$, we introduce an indeterminate x_{uv}. If G is directed, the x_{uv} variables are distinct for every edge (u,v). If G is undirected, then $(u,v) \in E$ if and only if $(v,u) \in E$, and we set $x_{uv} = x_{vu}$ for every edge (u,v).

Given a walk $W = \langle v_0,\ldots,v_r\rangle$ viewed as a sequence of vertices, we let the weight

$$(W) = \prod_{j=0}^{r \quad 1} x_{v_j v_{j+1}} \tag{56}$$

of W be the monomial recording all pairs of consecutive vertices traversed by W. By convention, the weight of a walk W of length zero (i.e., a single vertex) is $(W) = 1$.

Given a family of walks $P = \langle W_1,\ldots,W_r\rangle$, we assign it weight

$$(P) = \prod_{j=1}^{r} (W_j) \tag{57}$$

equal to the product of the weights monomials of the individual walks it contains. For the special case of pairs of paths $P = \langle P_1,P_2\rangle$ we also write

$$(P_1,P_2) = (P) = (P_1)\ (P_2) \tag{58}$$

for convenience.

Given a collection $\mathcal{F}$ of walks or families of walks in G, we say that a "polynomial F enumerates $\mathcal{F}$," or equivalently "F is the enumerating polynomial for $\mathcal{F}$," if

$$F = \sum_{S \in \mathcal{F}} \mathrm{wt}(S). \tag{59}$$

Edge-Based

In Section 7.2 we will design connectivity algorithms by enumerating edge-disjoint families of walks. For this task, it is more natural to view walks as sequences of edges rather than vertices, and assign weights which record pairs of adjacent edges in the walk instead of recording adjacent pairs of vertices. For this edge-based enumeration, we assume that G is directed, since this is the only context in which we will apply this enumeration.

For every pair of edges (e, f) in G such that $\mathrm{head}(e) = \mathrm{tail}(f)$ (i.e., edge e enters the vertex that edge f exits), we introduce an indeterminate variable x_{ef}. Given a walk

$$W = \langle e_1, \ldots, e_\ell \rangle,$$

viewed as a sequence of edges e_j, we let the weight

$$\mathrm{wt}(W) = \prod_{j=1}^{\ell-1} x_{e_j e_{j+1}} \tag{60}$$

of W be the monomial $\mathrm{wt}(W)$ recording all pairs of consecutive edges traversed by W. By convention, we assign a walk W of length one (i.e., a single edge) the weight $\mathrm{wt}(W) = 1$.

Given a family of walks $\mathcal{C} = \langle W_1, \ldots, W_r \rangle$, we still define the weight of this family

$$\mathrm{wt}(\mathcal{C}) = \prod_{i=1}^{r} \mathrm{wt}(W_i) \tag{61}$$

to be the product of the weights of the individual walks, and given a collection of $\mathcal{F}$ of walks or families of walks, we still define the enumerating polynomial for $\mathcal{F}$ to be the sum

$$\sum_{S \in \mathcal{F}} \mathrm{wt}(S)$$

of the weights of all members of $\mathcal{F}$.

6.3 Formal Power Series

In Chapter 7, our algorithms for computing graph connectivities work by enumerating families of disjoint walks. This enumeration involves handling infinite sums, which entails working with formal power series. To that end, in this section we review the definition and properties of power series. These results mostly involve observing that the basic facts which hold in the finite setting of polynomials still hold in the infinite setting of formal series.

Definitions

Fix a finite set J, and let $\{x_j\}$ indexed by $j \in J$ be the set of variables we work over. A polynomial is a finite linear combination of products of the x_j variables. A formal power series is a generalization of polynomials which allows for infinite sums.

Let $\mathbb{N}^J$ be the set of all sequences of nonnegative integers indexed by J. Given $\mathbf{d} \in \mathbb{N}^J$, we let d_j denote the j^{th} element of $\mathbf{d}$ for each $j \in J$. Then a *formal power series* F is identified by a sequence of coefficients $a_{\mathbf{d}}$ in $\mathbb{F}$, one for each $\mathbf{d} \in \mathbb{N}^J$, and we write

$$F = \sum_{\mathbf{d} \in \mathbb{N}^J} a_{\mathbf{d}} \prod_{j \in J} x_j^{d_j}.$$

We let $\mathbf{0}$ denote the all-zeros sequence in $\mathbb{N}^J$, and say $a_{\mathbf{0}}$ is the *constant term* of F. In general, given $\mathbf{d} \in \mathbb{N}^J$, the monomial corresponding to $\mathbf{d}$ in F (if it appears with nonzero coefficient $a_{\mathbf{d}} \neq 0$) is said to have degree

$$\sum_{j \in J} d_j.$$

Addition and Multiplication

Given formal series

$$F = \sum_{\mathbf{d} \in \mathbb{N}^J} a_{\mathbf{d}} \prod_{j \in J} x_j^{d_j} \quad \text{and} \quad H = \sum_{\mathbf{d} \in \mathbb{N}^J} b_{\mathbf{d}} \prod_{j \in J} x_j^{d_j}$$

we define their sum

$$F + H = \sum_{\mathbf{d} \in \mathbb{N}^J} (a_{\mathbf{d}} + b_{\mathbf{d}}) \prod_{j \in J} x_j^{d_j}$$

and product

$$F \cdot H = \sum_{\mathbf{d} \in \mathbb{N}^J} \left(\sum_{\substack{\mathbf{d_1}, \mathbf{d_2} \in \mathbb{N}^J \\ \mathbf{d_1} + \mathbf{d_2} = \mathbf{d}}} a_{\mathbf{d_1}} b_{\mathbf{d_2}} \right) \prod_{j \in J} x_j^{d_j} \tag{62}$$

in the natural way, generalizing arithmetic over polynomials (in the above equation, $\mathbf{d_1} + \mathbf{d_2} = \mathbf{d}$ means that $(d_1)_j + (d_2)_j = d_j$ for all $j \in J$). These operations make the set of polynomials over $\mathbb{F}$ a subring of the ring of formal power series, where the additive and multiplicative identities are the constant polynomials 0 and 1 respectively.

Inversion

Given a power series F, we say H is a *multiplicative inverse* of F if

$$F \cdot H = 1. \tag{63}$$

In order for the above equation to hold, the product of the constant terms of F and H must equal 1. In particular, F must have nonzero constant term to have a multiplicative inverse. The following fact shows that this condition is all that is needed for multiplicative inverses to exist, and that in fact multiplicative inverses are unique.

Proposition 6.2 (Power Series Inversion). Let F be a formal power series with nonzero constant term. Then there is a unique formal series H such that $F \cdot H = 1$.

Proof. Suppose

$$F = \sum_{\mathbf{d} \in \mathbb{N}^J} a_{\mathbf{d}} \prod_{j \in J} x_j^{d_j}.$$

We define the sequence $b_{\mathbf{d}}$ of coefficients in $\mathbb{F}$ for all $\mathbf{d} \in \mathbb{N}^J$ by taking

$$b_0 = (a_0)^{-1}$$

and then inductively setting

$$b_{\mathbf{d}} = -\left(\sum_{\mathbf{d}' < \mathbf{d}} a_{\mathbf{d}-\mathbf{d}'} b_{\mathbf{d}'} \right) \tag{64}$$

where $\mathbf{d}' < \mathbf{d}$ means that $\mathbf{d}' \in \mathbb{N}^J$ is componentwise less than or equal to $\mathbf{d}$, and $\mathbf{d}' \neq \mathbf{d}$ (and $(\mathbf{d} - \mathbf{d}')$ is the sequence with j^{th} term $(d_j - d'_j)$ for all $j \in J$).

By assumption $a_0 \neq 0$, so a_0 is invertible in $\mathbb{F}$, so b_0 is well-defined.

Then if we set

$$H = \sum_{\mathbf{d} \in \mathbb{N}^J} b_{\mathbf{d}} \prod_{j \in J} x_j^{d_j}$$

it follows from the definition of multiplication in eq. (62), the relationship from eq. (64), and the fact that $a_0 \cdot b_0 = 1$, that we have

$$F \cdot H = 1.$$

This inverse is unique, because for any formal series H' satisfying $F \cdot H' = 1$, we have

$$H' = H' \cdot 1 = H' \cdot (F \cdot H) = (H' \cdot F) \cdot H = 1 \cdot H = H.$$

Thus H is the unique multiplicative inverse of F as claimed.

Given a formal series F with nonzero constant term, we write $H = F^{-1}$ to denote the multiplicative inverse of F. This definition makes sense by Proposition 6.2.

Matrices of Power Series

In Chapter 7 we will work with *matrices* of formal series, because such objects arise naturally when computing inverses of polynomial matrices. The following formula will be useful for us to reason about the enumerative properties of certain matrices.

Proposition 6.3 (Geometric Series Formula). Let $\mathbf{X}$ be a square matrix with polynomial entries such that every entry of $\mathbf{X}$ has constant term zero. Then

$$(\mathbf{I} - \mathbf{X})^{-1} = \sum_{\ell=0}^{\infty} \mathbf{X}^{\ell}. \tag{65}$$

Proof. Since every entry of $\mathbf{X}$ has constant term zero, for any integer $\ell \geq 0$, the nonzero entries of $\mathbf{X}^\ell$ each have degree at least ℓ. Consequently, the infinite sum from the right-hand side of eq. (65) is well-defined, because for any $\mathbf{d} \in \mathbb{N}^J$, only finitely many terms contribute to the coefficient of

$$\prod_{j \in J} x_j^{d_j}$$

in each entry of the sum.

To prove the claim, it suffices to show that the product

$$(\mathbf{I} - \mathbf{X}) \left(\sum_{\ell=0}^{1} \mathbf{X}^\ell \right) \tag{66}$$

is equal to the identity matrix.

For each integer $d \geq 0$, let $\mathbf{M}_d$ be the matrix from eq. (66) with entries restricted to terms of degree at most d. Then since nonzero entries of $\mathbf{X}^\ell$ have degree at least ℓ, we see that $\mathbf{M}_d$ is equal to the matrix obtained by taking

$$(\mathbf{I} - \mathbf{X}) \left(\sum_{\ell=0}^{d} \mathbf{X}^\ell \right) = \sum_{\ell=0}^{d} \left(\mathbf{X}^\ell - \mathbf{X}^{\ell+1} \right) = \mathbf{I} - \mathbf{X}^{d+1}$$

and restricting to the terms with degree at most d. Since every nonzero entry of $\mathbf{X}^{d+1}$ has degree greater than d, we see that $\mathbf{M}_d = \mathbf{I}$ is the identity matrix. Since this equation holds for every $d \geq 0$, eq. (66) holds as well.

Chapter 7

Connectivity

7.1 Overview

Given a network, how can we measure how "connected" different parts of it are?

This question, which underlies many basic problems in graph theory, can be quantitatively answered using the notion of connectivity in graphs. Given a graph G with specified nodes s and t, the *connectivity* from s to t, denoted by $(s; t)$, is defined to be the maximum number of edge-disjoint s t paths in G. Connectivity is a well-studied concept, with many classical theorems in mathematics focusing on properties of graphs where every pair of vertices has high connectivity (e.g., see [Sch02, Chapter 15]). This concept is especially important in computer science, because the connectivity $(s; t)$ is equal to the *maximum flow* from s to t in the unweighted graph G, where we view edges as having capacity 1. Maximum flow is a foundational problem in combinatorial optimization, with numerous applications in graph algorithms and optimal transport, which further motivates studying connectivity.

The presence of connectivity in various areas of pure mathematics and computer science highlights the importance of designing fast algorithms for computing connectivities in graphs.

For fixed vertices s and t, the maximum flow from s to t can be computed in almost linear time [CKL+22]. This means that computing a single connectivity $(s; t)$ in a graph is easy—essentially optimal algorithms are known for this task.

In certain applications however, knowing the value of a single connectivity might not be so useful on its own. For example, if we have a large network where links between nodes can fail, and want to identify which clusters of nodes are likely to remain connected by paths in the network even in the presence of edge failures, it may more informative to know the values of *multiple* connectivities in the graph.

This motivates the All-Pairs Connectivity (APC) problem, where we are tasked with computing connectivities for all pairs of vertices in a given graph.

> **All-Pairs Connectivity (APC)**
>
> Given a graph G, compute $(s; t)$ for all pairs of vertices $(s; t)$ in G.

Suppose the input graph G has n vertices and m edges. Given such a graph, how quickly can we solve the APC problem?

In undirected graphs, it is known that APC can be solved in $O(n^2)$ time [AKL+22]. This runtime is near-optimal, since to solve APC we need to output connectivity values for each pair of vertices, which necessarily takes (n^2) time.

What about in general directed graphs? Well, a naive approach is to solve APC by computing $(s;t)$ separately for each pair of vertices $(s;t)$ using a fast maximum flow algorithm. Each maximum flow call can be computed in $m^{1+o(1)}$ time by [CKL+22], so this approach yields an $n^2m^{1+o(1)}$ time algorithm for APC. Despite the simplicity of this strategy, this naive algorithm is currently the fastest algorithm known for solving APC in dense directed graphs! In sparse graphs however, [CLL13] presented a faster algorithm for APC:

Theorem 7.1: All-Pairs Connectivity Algorithm

There is an algorithm solving APC on m-edge graphs in $O(m^{\ell})$ time.

In the Exact subsection of Section 7.2, we present a proof of Theorem 7.1, different from the original analysis of [CLL13], using the framework of [Akm24].

Why is computing multiple connectivities harder in directed graphs?

The current best algorithms for solving APC in undirected graphs involve constructing an object known as a Gomory-Hu tree, a data structure which succinctly represents all the connectivity information of the input graph [AKL+22]. There are examples of directed graphs which provably do not admit Gomory-Hu trees [Ben95]. Consequently, the main technique we currently have for designing fast algorithms for APC in undirected graphs does not generalize to the directed case, and this barrier arises from the fact that the structure of connectivities in directed graphs is more complex than the corresponding structure in undirected graphs.

Besides the limited applicability of current approaches, APC appears to be harder on directed graphs than undirected graphs because of various conditional lower bounds. For example, as we discuss in the Better Lower Bounds subsection of Section 7.4, various hardness hypotheses from fine-grained complexity imply that APC requires $n^{3\ o(1)}$ time to solve over general directed graphs [KT18, AGI+18]. This gives additional evidence that APC is inherently more difficult on directed graphs than on undirected graphs.

The Relaxation: Bounded Connectivity

The current lack of progress in obtaining faster algorithms for APC in general directed graphs naturally motivates looking at easier versions of this problem.

One way of constructing easier variants of APC is to relax the amount of information we are expected to return. Instead of computing $(s;t)$ exactly for example, we can instead ask that we merely return some useful information about $(s;t)$.

For example, consider the task of determining, for each pair of vertices $(s;t)$ in G, whether $(s;t)$ 1. From the definition of connectivity, this is equivalent to determining for each pair of vertices $(s;t)$, if G contains an s t path. This is the Transitive Closure problem, a classic computational problem in graph theory. Transitive Closure can be solved in $O(n^{\ell})$ time [FM71]. Moreover, if it was possible to solve Transitive Closure in $O(n^{\ell\ ''})$ time for

any constant $\varepsilon > 0$, then one could also solve the Boolean Matrix Multiplication problem (a certain variant of multiplying two $n \times n$ matrices with entries in $\{0, 1\}$) in $O(n^{\omega-\varepsilon})$ time as well [FM71, Theorem 3]. Currently, no algorithm for Boolean Matrix Multiplication is known that runs polynomially faster than the integer matrix multiplication runtime of $O(n^{\omega})$. Because of this, it has been hypothesized that Boolean Matrix Multiplication requires $n^{\omega - o(1)}$ time to solve. Under this hypothesis, the $O(n^{\omega})$ time algorithm for Transitive Closure is optimal.

We can interpolate between the Transitive Closure problem and the general APC problem by more generally checking for each pair of vertices (s, t) in G, how large $\lambda(s, t)$ is compared to a fixed threshold k. This is the task posed in the k-Bounded All-Pairs Connectivity problem:

> k-Bounded All-Pairs Connectivity (k-APC)
>
> Given a graph G, compute $\min(k, \lambda(s, t))$ for all pairs of vertices (s, t) in G.

The k-APC problem is relevant in contexts where knowing the precise connectivity values between "well-connected" nodes is not important, and instead we care more about distinguishing for each pair of vertices whether its connectivity is small or large (where k is our cutoff for what counts as "small" and "large"). k-APC recovers the Transitive Closure problem for $k = 1$, and recovers the general APC problem if we set k to be at least the maximum connectivity value between any pair of node in G (for example, if G is a simple graph, then $\lambda(s, t) \leq n - 1$ for all $s, t \in V$, so in this case k-APC recovers for APC for $k = n - 1$). Because of this k-APC should intuitively be easier for small k, and harder for large k.

How quickly, we we solve the k-APC problem?

When $k = 1$, we have already seen that k-APC can be solved in $O(n^{\omega})$ time, and this runtime is optimal under a plausible hardness hypothesis.

When $k = 2$, it is similarly known that k-APC can be solved in $\tilde{O}(n^{\omega})$ time [GGI+17].

However, already for $k = 3$ it was an open problem whether k-APC admitted any non-trivial algorithm! That is, it was not known how to solve 3-APC faster than solving the more general APC problem. In [AJ24, Theorem 4], we resolved this open problem, by proving the following result:

> **Theorem 7.2: k-Bounded All-Pairs Connectivity Algorithm**
>
> There is an algorithm solving k-APC on n-node graphs in $\tilde{O}((kn)^{\omega})$ time.

Theorem 7.2 shows that for *any constant* k, we can solve k-APC in $\tilde{O}(n^{\omega})$ time, essentially matching the hypothesized optimal $O(n^{\omega})$ runtime of 1-APC. In the Bounded subsection of Section 7.2, we prove Theorem 7.2, following [Akm24].

Vertex Connectivity

The connectivity $\lambda(s, t)$ is defined in terms of edge-disjoint paths. A natural variant of this measure, relevant in certain applications, arises if we work with *vertex-disjoint* paths instead. The *vertex connectivity* from s to t, denoted by $\kappa(s, t)$, is defined to be the maximum number of internally vertex-disjoint paths from s to t in G. Here "internally vertex-disjoint" means

that the paths are allowed to overlap at the endpoints s and t, but must not have any other common vertices.

Given this notion, we can define the k-Bounded All-Pairs Vertex Connectivity problem, the vertex connectivity analogue of the k-APC problem.

> k-Bounded All-Pairs Vertex Connectivity (k-APVC)
>
> Given a graph G, compute $\min(k;\ \ (s;t))$ for all pairs of vertices $(s;t)$ in G.

Can k-APVC be solved as quickly as k-APC? In [AJ24, Theorem 5] we showed that the answer to this question is yes, by proving the following result:

Theorem 7.3: k-Bounded All-Pairs Vertex Connectivity Algorithm

There is an algorithm solving k-APVC on n-node graphs in $O(k^2 n^{\ l})$ time.

In the All-Pairs subsection of Section 7.3, we present a proof of Theorem 7.3, different from the original analysis of [AJ24], using the framework of [Akm24].

Global Vertex Connectivity

Beyond the "all-pairs" problems we have seen so far, significant research has gone into "global" variants of vertex connectivity computation in graphs. Given a graph G, we define its vertex connectivity $\ (G)$ to be

$$(G) = \min_{s;t2V} \ (s;t):$$

Computing $\ (G)$ is interesting because it provides a single number which quantifies the robustness of connections in the network G. It is known that $\ (G)$ is equal to the smallest number of vertices which must be deleted to disconnect G (i.e., make it so that for some pair of vertices $(s;t)$, there is no $s \quad t$ path in the modified graph) whenever G is not a complete graph [Fra11, Theorem 2.5.26], so $\ (G)$ is a natural statistic of G to study. Beyond its connection to network reliability, many classical results in graph theory involving understanding properties of graphs G which satisfy $\ (G) \quad k$, for some fixed, small positive integers k. For example,

These connections within computer science and mathematics highlight the importance of algorithms for determining whether the vertex connectivity of a graph is smaller or large. This motivates the k-Vertex Connectivity problem, where we are given an integer $k \quad 1$, and are tasked with determining if $\ (G) \quad k$.

> k-Vertex Connectivity
>
> Given a graph G, determine if $\ (G) \quad k$.

A long line of research has studied algorithms for k-Vertex Connectivity (see e.g., the works listed in [CQ21, Table 2]). By combining the almost-linear maximum flow algorithm of [CKL+22] with [LNP+21, Theorem 1.2], it is now known that we can compute the exact

value of (G) in $n^{2+o(1)}$ time. Consequently, k-Vertex Connectivity can be solved for any k in $n^{2+o(1)}$ time, which is essentially optimal in dense graphs. Before this almost-quadratic time algorithm was achieved, the fastest algorithm for k-Vertex Connectivity in dense graphs, for small values of k, came from the following result of [CR94, Section 5]:

Theorem 7.4: k-Bounded Vertex Connectivity Algorithm

We can solve k-Vertex Connectivity on n-node graphs in $O(n^{!} + nk^{!})$ time.

Even though the $O(n^{!} + nk^{!})$ runtime from Theorem 7.4 is slower than the $n^{2+o(1)}$ runtime which we now have for k-Vertex Connectivity, the algorithm establishing Theorem 7.4 remains interesting because of the alternate techniques it employs. For example, we can prove Theorem 7.4 without applying almost-linear maximum flow algorithms, or relying on the fast data structures for "kernelization" used in [LNP+21].

The algorithm of [CR94] builds off a previous $O(n^{!} + kn^{!})$ time algorithm of [LLW88] for k-Vertex Connectivity on undirected graphs. The original proof in [LLW88] was motivated by a physical interpretation of connectivity, and how it relates to an equilibrium configuration of a certain convex embedding of the vertices of G. The proof in [CR94] similarly takes the perspective of convex embeddings, to reduce computing connectivities to certain Laplacian systems of equations.

In the Global subsection of Section 7.3, we present a simple proof of Theorem 7.4, different from the original analysis of [CR94], using the framework of [Akm24].

Graph Preliminaries

We use the definitions from Section 6.1. In particular, $G = (V; E)$ is the input graph, with $jVj = n$ nodes and $jEj = m$ edges. We assume that G is weakly connected (i.e., the underlying undirected graph of G is connected), so that $m \quad n \quad 1$. This is without loss of generality for each of the problems we consider, since for the "all-pairs" problems APC, k-APC, and k-APVC, we can compute connectivites between pairs of vertices separately on each weakly connected component; and for the k-Vertex Connectivity problem, any graph that is not weakly connected has $(G) = 0$, so the problem is trivial in this case.

Given a vertex s, we define $E_{\text{out}}(s)$ to be the set of edges exiting s, and $V_{\text{out}}(s)$ to be the set of out-neighbors of s. Similarly, given a vertex t, we define $E_{\text{in}}(t)$ to be the set of edges entering t, and $V_{\text{in}}(t)$ to be the set of in-neighbors of t. We let

$$V_{\text{out}}[s] = V_{\text{out}}(s) \; [\; fsg \quad \text{and} \quad V_{\text{in}}[t] = V_{\text{in}}[t] \; [\; ftg \tag{67}$$

be the closed out-neighborhood of s and closed in-neighborhood of t respectively. We write

$$\deg_{\text{out}}(s) = jE_{\text{in}}(s)j \quad \text{and} \quad \deg_{\text{in}}(t) = jE_{\text{out}}(t)j$$

to denote the indegree of s and outdegree of t respectively.

Matrix Preliminaries

We use bold font, such as $\mathbf{M}$, to denote matrices. Given a matrix $\mathbf{M}$, a row index i, and column index j, we let $\mathbf{M}[i; j]$ denote the $(i; j)$ entry of $\mathbf{M}$. Given subsets I and J of row

and column indices respectively, we let $\mathbf{M}[I; J]$ be the submatrix of $\mathbf{M}$ restricted to rows in I and columns in J. We also let $\mathbf{M}[I;]$ be the submatrix restricted to rows in I and all columns, and $\mathbf{M}[; J]$ be the submatrix on all rows and restricted to columns in J.

We let $\mathsf{rank}\,\mathbf{M}$ denote the rank of $\mathbf{M}$, defined to be largest nonnegative integer r such that $\mathbf{M}$ contains an $r \quad r$ submatrix with nonzero determinant. Equivalently, $\mathsf{rank}\,\mathbf{M}$ is the dimension of the image of $\mathbf{M}$. This second definition shows that the rank of a matrix cannot increase after multiplication with another matrix. When $\mathbf{M}$ is a square matrix, we let $\det \mathbf{M}$ denote the determinant of $\mathbf{M}$, $\mathsf{adj}(\mathbf{M})$ denote the adjoint of $\mathbf{M}$, and $\mathbf{M}^{\ 1}$ denote the inverse of $\mathbf{M}$ (if it exists). A matrix $\mathbf{M}$ is invertible if and only if its determinant is nonzero.

Matrix Computation

We recall the following standard results concerning matrix computation.

Proposition 7.5 (Matrix Inversion). For any positive integer a, we can compute the inverse of an $a \quad a$ matrix over a field in $O(a^{!})$ field operations.

See [BH74] for a sketch of the proof of Proposition 7.5. For a more modern proof of Proposition 7.5, see the lecture notes [SCWW21].

Proposition 7.6 (Matrix Rank). For any positive integers a and b, we can compute the rank of an $a \quad b$ matrix over a field in $O(ab^{!\ 1})$ field operations.

See [IMH82] for a proof of Proposition 7.6. For a discussion of more recent advances in algorithms for computing matrix rank, see [CKL13].

Rational Identity Testing

To prove correctness for our algorithms, we use the following extension of Proposition 6.1, which shows that the random evaluation of a nonzero rational function, whose numerator and denominator have low degree, over a large field is nonzero with high probability.

Corollary 7.7. Let P and Q be two nonzero r-variate polynomials, each of degree at most d. Let $R = P/Q$. Then a uniform random evaluation of R over F^r is nonzero with probability at least $1 \quad 2d/|\mathsf{F}|$.

Proof. Under random evaluation over F^r, by Proposition 6.1 and the union bound, P and Q are both nonzero with probability at least $1 \quad 2d/|\mathsf{F}|$. So with this probability, $R = P/Q$ also has nonzero evaluation, as claimed.

7.2 Edge Connectivity

In this section, we present algorithms for the APC and k-APC problems. We follow the strategy suggested by Chapter 6, and design our algorithms by obtaining polynomials which enumerate various collections of edge-disjoint paths in G.

Throughout this section, $\deg_{\mathsf{out}}(s)$ denotes the number of edges exiting a vertex s, and $\deg_{\mathsf{in}}(t)$ denotes the number of edges entering a vertex t.

Exact

In this subsection, we present an $O(m')$ time algorithm for APC and prove Theorem 7.1. This result was originally proved in [CLL13, Theorem 1.4]. We present an alternate exposition for this theorem, following [Akm24]. After establishing Theorem 7.1, in the next subsection we will show how to build off the ideas employed in our APC algorithm to solve the parameterized relaxation k-APC faster for small k.

Graph Assumptions (Exact Connectivity Case)

Throughout the current subsection (the Exact subsection of Section 7.2), we allow G to be a multigraph. That is, G is allowed to have multiple *parallel edges* between the same pair of vertices. We do assume however, that G does not have any self-loops. This is without loss of generality, since self-loops do not affect the connectivity between distinct pairs of vertices.

Field Size

We recall the preliminaries from the Finite Field Computation subsection of Section 6.1. In particular we work over a field $\mathsf{F} = \mathsf{F}_{2^q}$. For solving APC, we set q to be the smallest positive integer with

$$2^q \quad 2m^2n^3: \tag{68}$$

Note that since $n \quad 1 \quad m$, we have $q = \quad (\log m)$.

Enumerating Walks

To solve APC, we need to compute connectivity values. Since connectivity is defined to be the maximum number of edge-disjoint paths between two given vertices, we can compute connectivities by *enumerating* families of edge-disjoint walks between vertices in the input graph (following the general framework of Section 6.2). To enumerate such families of walks, it will be helpful to first learn how to enumerate individual walks in the graph. We will do this by working with a symbolic edge-adjacency matrix $\mathbf{X}$ for G.

For every pair of edges $(e; f)$ with $\mathsf{head}(e) = \mathsf{tail}(f)$ (i.e., edge e enters the vertex that edge f exits), we introduce an indeterminate variable x_{ef}. We will use these variables to enumerate families of walks in G, following the discussion from Edge-Based subsection of Section 6.2. In particular, each walk W is assigned a monomial (W) recording the consecutive pairs of edges it traverses as in eq. (60), and each collection $C = \langle W_1; : : : ; W_r \rangle$ of walks is assigned a monomial (C) according to eq. (61).

Let $\mathbf{X}$ be the $m \quad m$ matrix with rows and columns indexed by E such that for each pair of edges $(e; f)$ we have

$$\mathbf{X}[e; f] = \begin{cases} x_{ef} & \text{if } \mathsf{head}(e) = \mathsf{tail}(f) \\ 0 & \text{otherwise:} \end{cases} \tag{69}$$

Given edges $e; f \ 2 \ E$ and an integer $` \quad 1$, let $W_{`}(e; f)$ denote the set of all walks beginning at e and ending at f of length $`$. Equation (69) shows that the $(e; f)$ entry of $\mathbf{X}$ enumerates all walks of length two from e to f in G, i.e, the walks in $W_2(e; f)$. The next result observes that higher powers of X enumerate longer walks in G.

Proposition 7.8. For any edges $e; f \in E$ and integer $\ell \geq 0$, we have

$$\mathbf{X}^{\ell}[e; f] = \sum_{W \in W_{\ell+1}(e;f)} (W).$$

Proof. By expanding out the definition of matrix multiplication, we see that

$$\mathbf{X}^{\ell}[e; f] = \sum_{\substack{e_0,\ldots,e_\ell \in E \\ e_0 = e \\ e_\ell = f}} \prod_{j=0}^{\ell-1} \mathbf{X}[e_j; e_{j+1}].$$

By definition, $\mathbf{X}[e_j; e_{j+1}] = x_{e_j e_{j+1}}$ if we can step from e_j to e_{j+1} in G, and is zero otherwise. Thus, the product

$$\prod_{j=0}^{\ell-1} \mathbf{X}[e_j; e_{j+1}]$$

is nonzero if and only if $W = \langle e_0, \ldots, e_\ell \rangle$ is a walk of length $(\ell + 1)$ in G. In this case,

$$\prod_{j=0}^{\ell-1} \mathbf{X}[e_j; e_{j+1}] = \prod_{j=0}^{\ell-1} x_{e_j e_{j+1}} = (W)$$

so we have

$$\mathbf{X}^{\ell}[e; f] = \sum_{W \in W_{\ell+1}(e;f)} (W)$$

as claimed.

Corollary 7.9 (Enumerating Walks). For any edges $e; f \in E$, we have

$$(\mathbf{I} - \mathbf{X})^{-1}[e; f] = \sum_{\ell=0}^{\infty} \left(\sum_{W \in W_{\ell+1}(e;f)} (W) \right).$$

Proof. By eq. (69), every entry of $\mathbf{X}$ has constant term zero. So the claim follows by combining the geometric series formula from Proposition 6.3 with Proposition 7.8.

Enumerating Edge-Disjoint Families of Walks

Given subsets of edges $S; T \subseteq E$ of equal size $|S| = |T| = r \geq 1$ and an integer $\ell \geq 1$, we define $F_\ell(S; T)$ to be the family of collections of r walks of total length ℓ, beginning at different edges of S and ending at different edges of T. If we fix some ordering $e_1, \ldots, e_r$ of the edges in S, then we can view each element of $F_\ell(S; T)$ as a sequence of walks $\langle W_1, \ldots, W_r \rangle$ satisfying the properties that each W_i begins at e_i and ends at some edge of T, the W_i walks all end at distinct edges of T, and the sum of the lengths of the W_i walks is ℓ. We also define $D_\ell(S; T) \subseteq F_\ell(S; T)$ to be the family of collections of r *edge-disjoint* walks from S to T of total length ℓ.

Our goal is to design enumerating polynomials for the $D_{\cdot}(S;T)$ families. We will do this using determinants. Determinants are a natural tool in this context because of their classical enumerative properties. For example, the Lindström-Gessel-Viennot lemma [AZ18, Chapter 29] for counting paths in directed acyclic graphs and the "combinatorial algorithm" for the determinant from [MV97] showcase how determinants can be used to enforce disjointness conditions on paths and walks.

> **Idea 8** Use determinants to "sieve out" collections of intersecting walks and recover edge-disjoint collections of walks.

Define $\mathbf{\Gamma} = (\mathbf{I} \quad \mathbf{X})^{1}$ for convenience. By Corollary 7.9, the entries of $\mathbf{\Gamma}$ enumerate walks in G. We use this claim to prove that determinants of submatrices of $\mathbf{\Gamma}$ enumerate collections of walks in G, beginning and ending at different edges. This observation follows almost immediately from the definition of the determinant.

Lemma 7.10 (Arbitrary Walks)**.** For any equal-size subsets of edges $S; T \quad E$, we have

$$\det \mathbf{\Gamma}[S;T] = \sum_{\ell=1}^{1} \left(\sum_{C 2 F_{\cdot}(S;T)} (C) \right)$$

Proof. Let $S(S;T)$ be the set of all bijections from S to T.

By the definition of the determinant, we have

$$\det \mathbf{\Gamma}[S;T] = \sum_{2 S(S;T)} \prod_{e 2 S} \mathbf{\Gamma}[e; \quad (e)]: \tag{70}$$

Note that we do not include a factor for the sign of in the above equation, because we work over a field F of characteristic two.

By Corollary 7.9, for each $e 2 S$ we have

$$\mathbf{\Gamma}[e; \quad (e)] = \sum_{\ell=0}^{1} \left(\sum_{W 2 W_{\cdot+1}(e; \quad (e))} (W) \right): \tag{71}$$

Write $S = \{e_1; : : : ; e_r\}$, where $r = |S| = |T|$.

By multiplying eq. (71) over all choices of $e 2 S$, we have

$$\prod_{e 2 S} \mathbf{\Gamma}[e; \quad (e)] = \prod_{i=1}^{r} \left(\sum_{\ell=0}^{1} \left(\sum_{W 2 W_{\cdot+1}(e_i; \quad (e_i))} (W) \right) \right): \tag{72}$$

Now, let L be the set of all r-tuples $(\ell_1; : : : ; \ell_r)$ of positive integers summing to

$$\ell_1 + \quad + \ell_r = \ell:$$

If we expand out the product on the right-hand side of eq. (72) and group terms according to the total length of the walks they come from, we obtain

$$\prod_{i=1}^{r}\left(\sum_{\ell=0}^{1}\left(\sum_{W\in W_{\ell+1}(e_i,\ (e_i))}(W)\right)\right)=\sum_{\ell=1}^{1}\left(\sum_{\substack{(\ell_1,\ldots,\ell_r)\in L\\ W_i\in W_{\ell_i}(e_i,\ (e_i))}}\prod_{i=1}^{r}(W_i)\right):$$

To clarify the expression above: in the right-hand side of the above equation, the inner summation is over all choices of positive integers $\ell_1,\ldots,\ell_r$ which sum to ℓ, and choices of walks $W_1,\ldots,W_r$ where W_i is a walk of length ℓ_i from e_i to (e_i). This is simply the result of distributing the product over $i\in[r]$ on the left-hand side of the equation over the sum of walks of all possible lengths from e_i to (e_i).

By chaining the above equation together with eqs. (70) to (72), and interchanging summation, we get that

$$\det\Gamma[S,T]=\sum_{\ell=1}^{1}\left(\sum_{\in S(S,T)}\ \sum_{\substack{(\ell_1,\ldots,\ell_r)\in L\\ W_i\in W_{\ell_i}(e_i,\ (e_i))}}\prod_{i=1}^{r}(W_i)\right): \tag{73}$$

To simplify eq. (73), observe that for any choice of bijection $\in S(S,T)$, lengths $(\ell_1,\ldots,\ell_r)\in L$, and walks $W_i\in W_{\ell_i}(e_i,\ (e_i))$, the collection $\langle W_1,\ldots,W_r\rangle$ is a sequence of walks from S to T of total length ℓ. Conversely, any collection $C\in F_\ell(S,T)$ has walks whose lengths sum up to ℓ, and corresponds to a unique bijection $\in S(S,T)$, obtained by checking which starting edges in S are connected to which ending edges in T by walks in C.

Thus, the inner nested summation above is equivalent to a single sum over all collections of walks in $F_\ell(S,T)$. Since the weight of a collection $C=\langle W_1,\ldots,W_r\rangle$ is

$$(C)=\prod_{i=1}^{r}(W_i);$$

the discussion from the previous paragraph together with Equation (73) implies that

$$\det\Gamma[S,T]=\sum_{\ell=1}^{1}\left(\sum_{C\in F_\ell(S,T)}(C)\right)$$

which proves the desired result.

We now formalize the intuition from Idea 8 and argue that the determinant sieves out collections of intersecting walks, so that only edge-disjoint families of walks are included in its enumeration. Our reasoning is similar to the standard proof of the previously mentioned Lindström-Gessel-Viennot lemma [AZ18, Chapter 29].

Lemma 7.11 (Intersecting Walks Cancel). For any equal-size subsets of edges $S,T\subseteq E$ and integer $\ell\geq 1$, we have

$$\sum_{C\in F_\ell(S,T)}(C)=\sum_{C\in D_\ell(S,T)}(C):$$

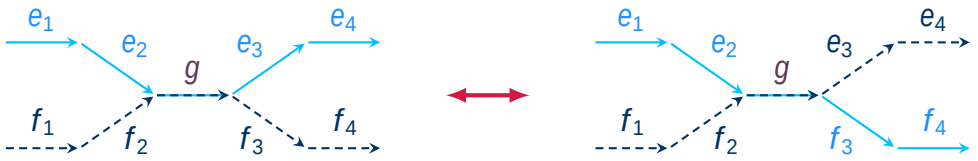

Figure 1: Given walks $W_i = \langle e_1; e_2; g; e_3; e_4 \rangle$ and $W_j = \langle f_1; f_2; g; f_3; f_4 \rangle$ overlapping at g, we can swap their suffixes to produce walks $W_i' = \langle e_1; e_2; g; f_3; f_4 \rangle$ and $W_j' = \langle f_1; f_2; g; e_3; e_4 \rangle$ which still overlap at g. The weights for both pairs $(W_i; W_j) = (x_{e_1e_2}x_{e_2g}x_{ge_3}x_{e_3e_4})\ (x_{f_1f_2}x_{f_2g}x_{gf_3}x_{f_3f_4})$ and $(W_i'; W_j') = (x_{e_1e_2}x_{e_2g}x_{gf_3}x_{f_3f_4})\ (x_{f_1f_2}x_{f_2g}x_{ge_3}x_{e_3e_4})$ agree because each pair traverses the same multiset of consecutive pairs of edges.

Proof. Fix $S; T \quad E$ and integer $` \quad 1$. Let $r = |S| = |T|$.

For convenience, abbreviate $F = F_`(S; T)$ and $D = D_`(S; T)$. Let $S = F \setminus D$ be the family of all collections of r walks beginning at different edges of S and ending at different edges of T, such that at least two walks in the collection intersect at an edge.

By the definition of S we have

$$\sum_{C \in F} (C) = \sum_{C \in D} (C) + \sum_{C \in S} (C):$$

So to prove the claim, it suffices to show that

$$\sum_{C \in S} (C)$$

is the zero polynomial. We prove this by pairing up collections C in S of equal weight (C), and observing that contributions from such collections vanish modulo two.

Fix an ordering $e_1; : : : ; e_r$ of the edges in S. Take any $C = \langle W_1; : : : ; W_r \rangle \in S$, with the walks ordered so that W_i begins at edge e_i. By assumption, at least two walks in C overlap at an edge. Let $i \in [r]$ be the smallest index such that W_i intersects some other walk in C at an edge. Let e be the first edge in W_i which is contained in another walk of C. Let $j \in [r]$ be the smallest index $j > i$ such that W_j overlaps with W_i at edge e.

We can split the walk W_i uniquely

$$W_i = A_i \quad B_i$$

as the concatenation of a prefix A_i not including edge e, and a suffix walk B_i which begins with edge e. We can similarly split W_j uniquely

$$W_j = A_j \quad B_j$$

as the concatenation of a prefix A_j not including e, and a suffix B_j beginning with e.

Now, define walks

$$W_i' = A_i \quad B_j \quad \text{and} \quad W_j' = A_j \quad B_i$$

by swapping the suffixes of W_i and W_j, as depicted in Figure 1. For all $l \in [r]$ with $l \notin \{i; j\}$, set $W_l' = W_l$. Define a new collection of walks

$$C' = \langle W_1'; : : : ; W_r' \rangle$$

by replacing W_i and W_j in C with W_i' and W_j' respectively.

Since W_i and W_j end at different edges of T, we know that $W_i' \neq W_i$ and $W_j' \neq W_j$. This shows that $C' \neq C$. Since the walks in C' still begin at different edges of S and end at different edges of T, $C' \in \mathcal{F}$. Also, W_i', W_j' overlap at an edge, so $C' \notin \mathcal{D}$. Thus $C' \in \mathcal{S}$.

Additionally, we claim that if we apply the above suffix swapping procedure (which we used to go from C to C') to the collection C', we recover C.

Indeed, for all $\ell \in [r]$ with $\ell < i$, the walk $W_\ell' = W_\ell$ does not intersect any other walk in C at an edge, by the definition of i. Since the multiset of edges traversed by walks in $C \setminus \{W_\ell\}$ and $C \setminus \{W_\ell'\}$ are the same, this means that W_ℓ' does not intersect any other walk in C' at an edge either. So i is also the smallest index in $[r]$ such that W_i' intersects some other walk in C at an edge. Since the prefixes of W_i' and W_i before edge e are the same, we see that e is also the first edge in W_i' which is contained in another walk of C'. Then because W_j' traverses edge e, and $W_\ell' = W_\ell$ for all $\ell \notin \{i, j\}$, we see that $j > i$ is the smallest index such that W_j' overlaps with W_i' at edge e. Then when we swap the suffixes of W_i' and W_j' after the first appearance of e on these walks, we recover W_i and W_j respectively, and so applying the suffix swapping procedure to C' produces the original collection C as claimed.

So, the suffix swapping routine described above partitions $\mathcal{S}$ into distinct pairs.

Suppose C and C' are paired up by the suffix swapping argument. Then C and C' traverse the same multiset of consecutive pairs of edges. Thus these collections

$$\mathrm{wt}(C) = \mathrm{wt}(C')$$

have the same weight. Since we work over a field of characteristic two, the above equation implies that each pair (C, C') of collections mapped to each other by suffix swapping satisfies

$$\mathrm{wt}(C) + \mathrm{wt}(C') = 0.$$

Since $\mathcal{S}$ is partitioned into such pairs, we have

$$\sum_{C \in \mathcal{S}} \mathrm{wt}(C) = 0.$$

Together with the discussion from the beginning of the proof, this proves the claim.

Corollary 7.12 (Edge-Disjoint Walks). For equal-size subsets of edges $S, T \subseteq E$, we have

$$\det \mathbf{\Gamma}[S, T] = \sum_{\ell=1}^{\infty} \left(\sum_{C \in \mathcal{D}_\ell(S,T)} \mathrm{wt}(C) \right).$$

Proof. This follows by combining Lemmas 7.10 and 7.11.

From Corollary 7.12, we see that determinants of submatrices of $\mathbf{\Gamma}$ produce enumerating polynomials for families consisting of collections of edge-disjoint walks. We want to use these polynomials to compute connectivities $\mathrm{wt}(s, t)$ for various vertices s and t. However, $\mathrm{wt}(s, t)$ is defined in terms of edge-disjoint *paths*, not walks. To next result will help us show that it is fine to work with walks instead of paths.

Lemma 7.13 (Edge-Disjoint Walks) Edge-Disjoint Paths). Let $S; T \quad E$ be subsets of edges of size $jSj = jTj = r$. If the graph G contains r edge-disjoint walks from S to T, then G also contains r edge-disjoint paths from S to T.

Proof. Let $hW_1; : : : ; W_r i$ be a collection of edge-disjoint walks from S to T in G. For each index $i \; 2 \; [r]$, let e_i and f_i be the first and last edges of W_i respectively. Note that under these definitions, we have $S = fe_1; : : : ; e_r g$ and $T = f f_1; : : : ; f_r g$.

For each $i \; 2 \; [r]$, let G_i be the subgraph of G including only the edges traversed by W_i. Let P_i be a shortest path from e_i to f_i in G_i. These paths are edge-disjoint, since they live in subgraphs on disjoint sets of edges. Thus $hP_1; : : : ; P_r i$ is a collection of r edge-disjoint paths from S to T in G, as desired.

Corollary 7.14. Let $S; T \quad E$ be subsets of edges of size $jSj = jTj = r$. Then $\det \Gamma[S; T]$ is a nonzero formal power series if and only if G contains r edge-disjoint paths from S to T.

Proof. Suppose $C = hP_1; : : : ; P_r i$ is a collection of r edge-disjoint paths from S to T in G. Then the term (C) occurs in the expansion of

$$\det \Gamma[S; T] \tag{74}$$

given by Corollary 7.12. Moreover, any collection of paths $C^0 \neq C$ from S to T has weight $(C^0) \neq (C)$, because C consists of edge-disjoint paths (so looking at the variables appearing in (C), we can recover C uniquely). Hence, no other term from the summation in Corollary 7.12 produces the same monomial (C). So (C) appears in eq. (74) with nonzero coefficient, which implies that the determinant from eq. (74) is a nonzero power series.

Suppose now that G does not contain r edge-disjoint paths from S to T. The contrapositive of Lemma 7.13 implies that G does not contain r edge-disjoint walks from S to T. Then Corollary 7.12 implies that eq. (74) is the zero polynomial. This proves the claim.

Random Evaluation

So far, we have constructed enumerating polynomials related to edge-disjoint walks using determinants. Following the approach suggested in Chapter 6, we next want to evaluate these polynomials at random points, in order to efficiently detect large collections of edge-disjoint walks. This will help us compute connectivity values.

We assign each pair of edges $(e; f)$ an independent, uniform random value a_{ef} in F.

Let $\mathbf{A}$ be the matrix obtained from $\mathbf{X}$ by evaluating each variable x_{ef} at a_{ef}. In other words, $\mathbf{A}$ is the random $m \quad m$ edge-adjacency matrix of G defined by taking

$$\mathbf{A} = \begin{cases} a_{ef} & \text{if } \mathsf{head}(e) = \mathsf{tail}(f) \\ 0 & \text{otherwise:} \end{cases}$$

Let $\mathbf{M} = (\mathbf{I} \quad \mathbf{A})^{\ 1}$ be the evaluation of $\mathbf{\Gamma}$ under this same random assignment.

Lemma 7.15. Let $S; T \quad E$ be subsets of edges of size $jSj = jTj = r$. If G contains r edge-disjoint paths from S to T, then $\det \mathbf{M}[S; T]$ is nonzero with probability at least $1 \quad 1=n^3$. If instead G does not have r edge-disjoint paths from S to T, then $\det \mathbf{M}[S; T]$ is zero.

Proof. By Corollary 7.14, the determinant

$$\det \mathbf{\Gamma}[S; T] \tag{75}$$

is a nonzero power series if and only if G contains r edge-disjoint paths from S to T.

So if G does not contain r edge-disjoint paths from S to T, then eq. (75) is the zero polynomial, so its random evaluation

$$\det \mathbf{M}[S; T]$$

must equal zero as claimed.

Otherwise, G contains r edge-disjoint paths from S to T, and eq. (75) is a nonzero power series. By the formula for the inverse of a matrix, we have

$$\mathbf{\Gamma}[S; T] = \frac{(\operatorname{adj}(\mathbf{I} \quad \mathbf{X}))[S; T]}{\det(\mathbf{I} \quad \mathbf{X})}. \tag{76}$$

The matrix $\mathbf{I} \quad \mathbf{X}$ has ones across its main diagonal, and every other entry of this matrix has constant term zero Then by the formula for the determinant of a matrix, $\det(\mathbf{I} \quad \mathbf{X})$ is a polynomial with constant term 1, so by Proposition 6.2 the multiplicative inverse of $\det(\mathbf{I} \quad \mathbf{X})$ is a well-defined power series. As a consequence, eq. (76) can be viewed as an equality between matrices of formal power series.

For convenience, write $Q = \det(\mathbf{I} \quad \mathbf{X})$.

Since S and T are sets of size r, by linearity of the determinant we have

$$\det \mathbf{\Gamma}[S; T] = \frac{\det(\operatorname{adj}(\mathbf{I} \quad \mathbf{X}))[S; T]}{Q^r}. \tag{77}$$

By assumption, the left-hand side of eq. (77) is nonzero. Consequently, the numerator

$$\det(\operatorname{adj}(\mathbf{I} \quad \mathbf{X}))[S; T]$$

on the right-hand side of eq. (77) must be a nonzero polynomial. Since each entry of $\mathbf{X}$ has degree at most 1, each entry of $\operatorname{adj}(\mathbf{I} \quad \mathbf{X})$ has degree less than m, so the numerator polynomial from the above equation has total degree less than rm. Similarly, in the previous discussion we observed that Q is a polynomial with constant term 1, so the denominator

$$Q^r = (\det(\mathbf{I} \quad \mathbf{X}))^r$$

of the right hand side of eq. (77) has constant term 1, and is thus a nonzero polynomial as well. Since each entry of X has degree at most 1, this denominator has degree at most rm.

So the right hand side of eq. (77) is the ratio of two nonzero polynomials, each with degree at most $rm \quad m^2$. Then by Corollary 7.7 and eq. (68), the random evaluation $\det \mathbf{M}[S; T]$ is nonzero over F with probability at least

$$1 \quad 2m^2/(2^q) \quad 1 \quad 1/n^3$$

as desired.

Algorithm 5. The All-Pairs Connectivity Algorithm

Inputs: A directed graph G.
Returns: The connectivity $(s;t)$ for each pair of vertices $(s;t)$ in G.

1. Compute $\mathbf{M} = (\mathbf{I} \quad \mathbf{A})^{1}$.
2. For each pair of vertices $(s;t)$, return

$$\operatorname{rank} \mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)]$$

as the value for $(s;t)$.

Lemma 7.16 (Connectivity via Rank). With high probability, for all $s;t \in V$ we have

$$(s;t) = \operatorname{rank} \mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)]:$$

Proof. Fix a pair of vertices $(s;t)$. Abbreviate $= (s;t)$. By Lemma 7.15 and the definition of connectivity, with probability at least $1 \quad 1=n^3$, 0 is the largest integer such that there exist subsets $S \quad E_{\text{out}}(s)$ and $T \quad E_{\text{in}}(t)$ of size with the property that $\det \mathbf{M}[S;T]$ is nonzero. By definition, this is the rank of $\mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)]$, so

$$= \operatorname{rank} \mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)]$$

with probability at least $1 \quad 1=n^3$ for our fixed pair of vertices $(s;t)$. So by the union bound over n^2 pairs of vertices, with probability at least $1 \quad 1=n$ we have

$$(s;t) = \operatorname{rank} \mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)]$$

for all pairs of vertices $(s;t)$ in G, as desired.

Lemma 7.16 establishes a direct connection between ranks of submatrices of $\mathbf{M}$ and connectivities in G. We leverage this connection to design Algorithm 5 solving APC.

Proof of Theorem 7.1. By Lemma 7.16, Algorithm 5 solves APC correctly. To prove the theorem, it remains to show we can implement Algorithm 5 to run in $O(m^{!})$ time.

Step 1 of Algorithm 5 takes $O(m^{!})$ time by Proposition 7.5, because we just need to invert the $m \quad m$ matrix $(\mathbf{I} \quad \mathbf{A})$.

In step 2 of Algorithm 5, for each pair of vertices $(s;t)$, we need to compute the rank of a matrix with dimensions $\deg_{\text{out}}(s) \quad \deg_{\text{in}}(t)$ matrix. By Proposition 7.6, this takes

$$\sum_{s;t \in V} \deg_{\text{out}}(s)(\deg_{\text{in}}(t))^{! \ 1} \tag{78}$$

field operations asymptotically. For each pair of vertices $(s;t)$, we have

$$\deg_{\text{out}}(s)(\deg_{\text{in}}(t))^{! \ 1} = (\deg_{\text{in}}(t))^{! \ 2} \ \deg_{\text{out}}(s) \deg_{\text{in}}(t) \quad m^{! \ 2} \ \deg_{\text{out}}(s) \deg_{\text{in}}(t):$$

By substituting this inequality into eq. (78), and observing that

$$\sum_{s2V} \deg_{\text{out}}(s) = \sum_{t2V} \deg_{\text{in}}(t) = m$$

we get that

$$\sum_{s;t2V} \deg_{\text{out}}(s)(\deg_{\text{in}}(t))^{\,!\ 1} \quad \sum_{s;t2V} m^{\,!\ 2}\ \deg_{\text{out}}(s)\deg_{\text{in}}(t) = m^{\,!\ 2}m^2 = m^{\,!}:$$

So step 2 of Algorithm 5 takes $O(m^{\,!})$ time.

So we can solve APC in $O(m^{\,!})$ time as claimed.

Bounded

In this subsection, we present an $O((kn)^{\,!})$ time algorithm for k-APC and prove Theorem 7.2. We will do this by building off the ideas that went into designing Algorithm 5 for solving the unparameterized APC problem in $O(m^{\,!})$.

Graph Assumptions (Exact Connectivity Case)

Throughout the current subsection (the Bounded subsection of Section 7.2), we allow G to be a multigraph. That is, G is allowed to have multiple *parallel edges* between the same pair of vertices. We assume however, that for any distinct vertices s and t, there are at most k edges from s to t. This is without loss of generality when solving the k-APC problem, since if there were more than k parallel edges from s to t, we could delete some and bring the count of parallel edges down to k without changing the value of $\min(k;\ (s;t))$. We additionally assume that G does not have any self-loops. This is also without loss of generality, since self-loops do not affect the connectivity between distinct pairs of vertices. These assumptions imply that $m \quad kn^2$.

Field Size

We recall the preliminaries from the Finite Field Computation subsection of Section 6.1. In particular we work over a field $F = F_{2^q}$. For solving k-APC, we set q to be the smallest positive integer with

$$2^q \quad 4k(m + 2kn)n^3: \tag{79}$$

Note that since $n \quad 1 \quad m$ and $k \quad m$, we have $q = \ (\log m)$.

Adapting the All-Pairs Connectivity Algorithms

Algorithm 5 solves APC by

1. inverting an $m \quad m$ matrix to produce a special matrix encoding connectivity information, and then

2. computing ranks of submatrices of the special matrix, whose dimensions are based off degrees of nodes in G, to compute connectivity values.

We want to adapt this strategy to solve k-APC in $O((kn)^{!})$ time. However, merely writing down the matrices used in Algorithm 5 already takes (m^2) time. To design an algorithm running in only $O((kn)^{!})$ time, we need to modify the steps of Algorithm 5 to work with much *smaller matrices*, perhaps with at most $O(kn)$ rows and columns.

Step 2 of Algorithm 5 (corresponding to item 2 above) is slow because vertices in G can have large degrees, which means that Algorithm 5 needs to compute ranks of large submatrices. To speed this up, we would ideally like to *reduce degrees* in G, while preserving the values of small connectivities.

> **Idea 9** If we only need to compute $(s; t)$ when this value is less than k, it should suffice to compute this connectivity in a modified *low-degree* graph where both $\deg_{\text{out}}(s)$ and $\deg_{\text{in}}(t)$ are at most k.

If we can reduce the degrees of vertices in G to at most k for example, then in item 2 above we would only need to compute ranks of k k submatrices for each pair of verticces, which we could do in the desired $O((kn)^{!})$ time bound.

Step 1 of Algorithm 5 (corresponding to item 1 above) seems to trickier to accelerate. Beyond the fact that the special matrix used in Algorithm 5 has m^2 entries, the construction of enumerating polynomials for collections of edge-disjoint paths involved variables x_{ef} for each pair of edges $(e; f)$ with $\text{head}(e) = \text{tail}(f)$. Even using the assumption that each pair of vertices in G has at most k parallel edges between them, G can contain (kmn) such pairs of edges, which is a bottleneck for the time it takes to evaluate any enumerating polynomial we construct on the x_{ef} variables.

To solve APC, we needed all the x_{ef} variables to be present in the enumerating polynomials, so that the monomial (C) for any collection of edge-disjoint paths C uniquely recorded the multiset of pairs of edges traversed by each path in C. This unique encoding property was essential to make sure that distinct collections of edge-disjoint paths C received distinct monomials (C), and we did not get any unwanted cancellations of terms.

When solving k-APC however, we only need to enumerate up to k edge-disjoint paths in a solution. Each path in G contains at most n vertices, and so a collection of k edge-disjoint paths in G uses fewer than kn edges. This suggests that for the purpose of enumerating up to k edge-disjoint paths, we might be able to employ a *less expressive enumeration* which involves fewer variables, yet still is enough to solve k-APC.

> **Idea 10** If we only need to detect up to k edge-disjoint paths, then since each path has at most n vertices, we should be able to enumerate using fewer variables.

Idea 10 suggests a way to decrease the number of variables in our enumerating polynomials, and thereby avoid the polynomial evaluation bottleneck. But how can we avoid the matrix size bottleneck of (m^2) discussed before?

Why did we need $\mathbf{M}$ in Algorithm 5 to be an m m matrix in the first place? Well, we wanted ranks of submatrices of $\mathbf{M}$ to be correspond to connectivity values. Since connectivities can be as large as m, we need $\mathbf{M}$ to be at least m m just so it could have submatrices

whose rank was m. When solving k-APC however, we only need to output connectivities with value at most k. This suggests that for solving k-APC, we can impose more structure on the matrix $\mathbf{M}$ which encodes connectivity values, and perhaps this structure will let us avoid explicitly using an $m \quad m$ matrix.

> **Idea 11** Since we only need to output small connectivity values in the k-APC problem, we should be able to get away with encoding connectivities in a *low-rank* matrix.

In the next parts, we try and formalize the intuition from Ideas 9 to 11.

Reducing Degrees

Recall that G is the input graph on n nodes and m edges. Based off Idea 9, we modify G to create a new graph G_{new}, which preserves the k-bounded connectivity information of G, yet will have the nice property that all relevant vertices have indegree and outdegree k.

For each vertex $v \ 2 \ V$, we introduce two new nodes v_{in} and v_{out}. Then we replace each edge $(u; v) \ 2 \ E$ with an edge $(u_{\text{out}}; v_{\text{in}})$. For each $v \ 2 \ V$, we also include k parallel edges from v to v_{out}, and k parallel edges from v_{in} to v. Let G_{new} be the new graph constructed in this way, and let V_{new} and E_{new} be its vertex and edge sets respectively.

By construction, G_{new} has $n_{\text{new}} = jV_{\text{new}}j = 3n$ nodes and $m_{\text{new}} = jE_{\text{new}}j = m + 2kn$ edges.

We refer to the nodes in $V \quad V_{\text{new}}$ which were originally in G as the *original vertices*. For $s; t \ 2 \ V$, we still let $(s; t)$ denote the connectivity from s to t in the original graph G. For $s; t \ 2 \ V_{\text{new}}$, we let $_{\text{new}}(s; t)$ denote the connectivity from s to t in G_{new}.

For the rest of Section 7.2, we let $E_{\text{out}}(s)$ and $E_{\text{in}}(t)$ denote the sets of edges exiting s and entering t in G_{new}.

Lemma 7.17 (Preserving Small Connectivities). For any original vertices $s; t \ 2 \ V$, we have

$$_{\text{new}}(s; t) = \min(k; \ (s; t)):$$

Proof. Fix $s; t \ 2 \ V$. Given an $s \quad t$ path P^0 in G_{new}, we recover a unique $s \quad t$ path P in G by looking at the sequence of original vertices P^0 passes through. Using this construction, any collection of r edge-disjoint paths from s to t in G_{new} recovers a collection of r edge-disjoint paths from s to t in G. This implies that $_{\text{new}}(s; t) \quad (s; t)$.

Since s has outdegree k in G_{new}, we also necessarily have $_{\text{new}}(s; t) \quad k$.

Thus the connectivity from s to t in G_{new} is at most $\min(k; \ (s; t))$.

Set $= \min(k; \ (s; t))$. The discussion so far shows that

$$_{\text{new}}(s; t) \quad : \tag{80}$$

By definition of , G contains edge-disjoint $s \quad t$ paths $P_1; : : : ; P$.

For each $i \ 2 \ [\ ]$, let P_i^0 be the $s \quad t$ path in G_{new} which passes through the same sequence of original vertices as P_i, and includes, for each edge $(u; v)$ in P_i, the i^{th} parallel edge from u_{out} to v_{in} (with respect to some fixed ordering among all the parallel edges). This is possible since k. Since the P_i are edge-disjoint, the P_i^0 are edge-disjoint as well.

So G_{new} has edge-disjoint $s \quad t$ paths, implying that

$$_{\text{new}}(s; t) \quad : \tag{81}$$

By eqs. (80) and (81), we have $_{\text{new}}(s; t) = \min(k; \ (s; t))$ as claimed.

Figure 2: When we substitute $x_{ef} = y_{e1}z_{1f} + \quad + y_{ek}z_{kf}$ (pictured here for $k = 3$) into X, we get the "simpler" matrix YZ. While powers of X enumerate walks in G, powers of YZ intuitively enumerate walks in a modified graph where after traversing an edge $e = (u; v)$, we have k di□erent versions of v we can choose to go to. The y_{ej} and z_{jf} variables in this enumeration only keep track of the individual edges traversed and versions of vertices we pick, instead of recording all pairs of consecutive edges traversed like the x_{ef} variables. This simpler enumeration su□ces to solve k-APC.

Low-Rank Enumeration

We redefine the matrix $\mathbf{X}$ from eq. (69) with respect to the new graph G_{new}, so that now rows and columns of $\mathbf{X}$ are indexed by edges in E_{new}.

We similarly redefine the matrix $\mathbf{\Gamma} = (\mathbf{I} \quad \mathbf{X})^{\ 1}$ with respect to G_{new}.

Motivated by Idea 10, we introduce some new variables.

For each pair $(e; j) \ 2 \ E_{\text{new}} \quad [k]$ we introduce an indeterminate y_{ej}.

Similarly, for each pair $(j; f) \ 2 \ [k] \quad E_{\text{new}}$ we introduce an indeterminate z_{jf}.

We define the $m_{\text{new}} \quad kn_{\text{new}}$ matrix $\mathbf{Y}$ by setting

$$\mathbf{Y}[e; (v; j)] = \begin{cases} y_{ej} & \text{if } \mathsf{head}(e) = v \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, we define the $kn_{\text{new}} \quad m_{\text{new}}$ matrix $\mathbf{Z}$ by setting

$$\mathbf{Z}[(v; j); f] = \begin{cases} z_{jf} & \text{if } \mathsf{tail}(f) = v \\ 0 & \text{otherwise.} \end{cases}$$

These matrices are defined so that under the variable substitution

$$x_{ef} = \sum_{j=1}^{k} y_{ej} z_{jf}$$

the matrix $\mathbf{X}$ simplifies to the *low-rank* matrix $\mathbf{YZ}$, as depicted in Figure 2.

Previously in Corollary 7.14, we characterized the existence of edge-disjoint paths in G via nonzero determinants of submatrices of $\mathbf{\Gamma}$. The following result shows that a similar characterization holds when we replace $\mathbf{X}$ with $\mathbf{YZ}$, provided we only care about enumerating up to k edge-disjoint paths.

Replacing the x_{ef} variables with the y_{ej} and z_{jf} variables is our way of implementing the intuition of Idea 10, and replacing $\mathbf{X}$ with $\mathbf{YZ}$ is our way of implementing Idea 11.

Lemma 7.18. Let $S; T \quad E_{\text{new}}$ be subsets of edges with size $jSj = jTj = r \quad k$. Then

$$\det(\mathbf{I} \quad \mathbf{YZ})^{\ 1}[S; T]$$

is a nonzero formal power series if and only if G_{new} has r edge-disjoint paths from S to T.

Proof. Suppose G does not contain r edge-disjoint paths from S to T. Then by Corollary 7.14, the determinant $\det \mathbf{\Gamma}[S;T]$ is identically zero as a power series. Consequently, $\det \mathbf{\Gamma}[S;T]$ remains zero even if we make the variable substitution

$$x_{ef} = \sum_{j=1}^{k} y_{ej} z_{jf}. \tag{82}$$

Under this substitution, the matrix $\mathbf{X}$ simplifies to $\mathbf{YZ}$. Since $\mathbf{\Gamma} = (\mathbf{I} \quad \mathbf{X})^{-1}$, we get that

$$\det (\mathbf{I} \quad \mathbf{YZ})^{-1}[S;T]$$

is the zero polynomial, as claimed.

Otherwise, G does contain r edge-disjoint paths from S to T.

In this case, Corollary 7.12 implies that

$$\det \mathbf{\Gamma}[S;T] = \sum_{\ell=1}^{1} \left(\sum_{C \in \mathcal{D}(S;T)} w(C) \right). \tag{83}$$

For each collection of walks C, let $\tilde{w}(C)$ be the monomial resulting from substituting eq. (82) into the monomial $w(C)$. This substitution turns $\mathbf{X}$ into $\mathbf{YZ}$, so we have

$$\det (\mathbf{I} \quad \mathbf{YZ})^{-1}[S;T] = \sum_{\ell=1}^{1} \left(\sum_{C \in \mathcal{D}(S;T)} \tilde{w}(C) \right). \tag{84}$$

Let $P = \langle P_1, \ldots, P_r \rangle$ be a collection of edge-disjoint paths from S to T in G.

For each $i \in [r]$, let E_i be the set of consecutive pairs of edges (e,f) traversed by P_i.

Then since

$$w(P) = \prod_{i=1}^{r} w(P_i) = \prod_{i=1}^{r} \prod_{(e,f) \in E_i} x_{ef}$$

by the definition of the weight of a collection of paths, we get that

$$\tilde{w}(P) = \prod_{i=1}^{r} \prod_{(e,f) \in E_i} \left(\sum_{j=1}^{k} y_{ej} z_{jf} \right).$$

If we expand the product on the right-hand side of the above equation, we see that one of the monomials produced is of the form

$$\prod_{i=1}^{r} \prod_{(e,f) \in E_i} y_{ei} z_{if}. \tag{85}$$

Note that in this step, we are using the fact that $r \leq k$.

Because P is a collection of edge-disjoint paths, the variables appearing in the monomial from eq. (85) allow us to uniquely recover P. Specifically, given the monomial from eq. (85),

Figure 3: Given edge-disjoint paths $P_1 = he_1; f_1i$ and $P_2 = he_2; f_2i$ in G, the determinant of the matrix $\Gamma[fe_1; e_2g; ff_1; f_2g]$ enumerates this pair via the monomial $!(P_1; P_2) = x_{e_1f_1} x_{e_2f_2}$. The variables in this monomial provide enough information to uniquely recover P_1 and P_2. In contrast, for $k = 2$, the determinant of $(\mathbf{I} \quad \mathbf{YZ})^{-1}[fe_1; e_2g; ff_1; f_2g]$ assigns this pair weight

$$\tilde{}(P_1; P_2) = (y_{e_1 1} z_{1f_1} + y_{e_1 2} z_{2f_1})(y_{e_2 1} z_{1f_2} + y_{e_2 2} z_{2f_2}):$$

One of the terms in the expansion of the above product is $y_{e_1 1} z_{1f_1} \; y_{e_2 2} z_{2f_2}$. We can read this term as saying "the first path P_1 traverses e_1 and f_1, and the second path P_2 traverses e_2 and f_2." So this monomial provides enough information to recover the pair of paths $hP_1; P_2i$ as well.

for each $i \in [r]$, the edges e for which the variable y_{ei} appears in the monomial are precisely the edges in P_i. Since P_i is a simple path beginning at a node of S and ending at node of T, given these edges we can recover their order P_i as well. So reading out the variables appearing in eq. (85) lets us identify P_i for each $i \in [r]$, and thus the collection P, as claimed.

See Figure 3 for an example of this unique recovery property in the case of $k = 2$.

Thus the monomial from eq. (85) appears with coefficient 1. Hence

$$\det(\mathbf{I} \quad \mathbf{YZ})^{-1}[S; T]$$

is a nonzero formal power series as claimed.

Thanks to Lemma 7.18, we have polynomials which can help us detect up to k edge-disjoint walks using determinants. However, the matrix $(\mathbf{I} \quad \mathbf{YZ})$ used in Lemma 7.18 has dimensions $m_{\text{new}} \quad m_{\text{new}}$, so working directly with it is not useful for obtaining a $O((kn)^{!})$ time algorithm. The following lemma helps us get around this, by leveraging the fact that $\mathbf{YZ}$ has low rank to relate the inverse of $(\mathbf{I} \quad \mathbf{YZ})$ to the inverse of a smaller matrix. This is the main technical reason that Idea 11 is useful for solving k-APC.

Lemma 7.19 (Geometric Series Identity). We have

$$(\mathbf{I} \quad \mathbf{YZ})^{-1} = \mathbf{I} + \mathbf{Y}(\mathbf{I} \quad \mathbf{ZY})^{-1}\mathbf{Z}:$$

Proof. By definition, every entry of $\mathbf{Y}$ and $\mathbf{Z}$ has constant term zero. This means that all the entries of $\mathbf{YZ}$ and $\mathbf{ZY}$ also have constant term zero.

Then by the geometric series formula of Proposition 6.3 we have

$$(\mathbf{I} \quad \mathbf{YZ})^{-1} = \mathbf{I} + (\mathbf{YZ}) + (\mathbf{YZ})^2 + \quad = \mathbf{I} + \mathbf{Y}\left(\sum_{`=0}^{1} (\mathbf{ZY})^{`}\right)\mathbf{Z}: \tag{86}$$

Applying Proposition 6.3 again, we have

$$\sum_{`=0}^{1} (\mathbf{ZY})^{`} = (\mathbf{I} \quad \mathbf{ZY})^{-1}:$$

Substituting the above equation into the rightmost side of eq. (86) yields

$$(\mathbf{I} \quad \mathbf{YZ})^{1} = \mathbf{I} + \mathbf{Y}(\mathbf{I} \quad \mathbf{ZY})^{1}\mathbf{Z}$$

as desired.

Lemma 7.19 is useful because it reduces computing the inverse of $(\mathbf{I} \quad \mathbf{YZ})$, an $m_{\text{new}} \quad m_{\text{new}}$ matrix, to computing the inverse of $(\mathbf{I} \quad \mathbf{ZY})$, a $kn_{\text{new}} \quad kn_{\text{new}}$ matrix.

Now, define subsets of edges

$$E_{\text{out}} = \bigcup_{s2V} E_{\text{out}}(s) \quad \text{and} \quad E_{\text{in}} = \bigcup_{t2V} E_{\text{in}}(t):$$

Set $\widetilde{\mathbf{Y}} = \mathbf{Y}[E_{\text{out}}; \ ]$ and $\widetilde{\mathbf{Z}}[\ ; E_{\text{in}}]$. Since every original vertex has outdegree and indegree exactly k in G_{new}, $\widetilde{\mathbf{Y}}$ is a $kn \quad kn_{\text{new}}$ matrix, and $\widetilde{\mathbf{Z}}$ is a $kn_{\text{new}} \quad kn$ matrix.

Random Evaluation

Following the approach suggested in Chapter 6, to detect edge-disjoint paths we now define random evaluations of the matrices $\mathbf{Y}$ and $\mathbf{Z}$ we have constructed.

For all pairs $(e; j)\ 2\ E_{\text{new}} \quad [k]$ and $(j; f)\ 2\ [k] \quad E_{\text{new}}$ we introduce independent, uniform random values b_{ej} and d_{jf} respectively over F.

Let $\mathbf{B}$ and $\mathbf{C}$ be the matrices obtained by setting

$$y_{ej} = b_{ej} \quad \text{and} \quad z_{jf} = c_{jf} \tag{87}$$

in $\mathbf{Y}$ and $\mathbf{Z}$ respectively, for all $(e; j)\ 2\ E_{\text{new}} \quad [k]$ and $(j; f)\ 2\ [k] \quad E_{\text{new}}$.

So $\mathbf{B}$ is the $m_{\text{new}} \quad kn_{\text{new}}$ matrix defined by setting

$$\mathbf{B}[e; (v; f)] = \begin{cases} b_{ej} & \text{if } \mathsf{head}(e) = v \\ 0 & \text{otherwise} \end{cases}$$

and $\mathbf{C}$ is the $kn_{\text{new}} \quad m_{\text{new}}$ matrix defined by setting

$$\mathbf{C}[(v; j); f] = \begin{cases} c_{jf} & \text{if } \mathsf{tail}(f) = v \\ 0 & \text{otherwise:} \end{cases}$$

Lemma 7.20. Let $S; T \quad E_{\text{new}}$ be subsets of edges of size $jSj = jTj = r \quad k$. If G_{new} contains r edge-disjoint paths from S to T, then

$$\det(\mathbf{I} \quad \mathbf{BC})^{1}[S; T]$$

is nonzero with probability at least $1 \quad 1=n^3$. If instead G_{new} does not have r-edge disjoint paths from S to T, then

$$\det(\mathbf{I} \quad \mathbf{BC})^{1}[S; T] = 0:$$

Proof. By Lemma 7.18, the formal power series

$$\det(\mathbf{I} \quad \mathbf{YZ})^{-1}[S;T] \tag{88}$$

is nonzero if and only if G_{new} contains r edge-disjoint paths from S to T.

So if G_{new} does not have r edge-disjoint paths from S to T, then eq. (88) is the zero polynomial, so its random evaluation

$$\det(\mathbf{I} \quad \mathbf{BC})^{-1}[S;T]$$

must equal zero as claimed.

Otherwise, G_{new} contains r edge-disjoint paths from S to T, and eq. (88) is a nonzero power series. By the formula for the inverse of a matrix, we have

$$(\mathbf{I} \quad \mathbf{YZ})^{-1}[S;T] = \frac{(\text{adj}(\mathbf{I} \quad \mathbf{YZ}))[S;T]}{\det(\mathbf{I} \quad \mathbf{YZ})}. \tag{89}$$

The matrix $(\mathbf{I} \quad \mathbf{YZ})$ has ones along its main diagonal, and every other entry of this matrix has constant term zero. Then by the formula for the determinant, $\det(\mathbf{I} \quad \mathbf{YZ})$ is a polynomial with constant term 1, so by Proposition 6.2 the multiplicative inverse of $\det(\mathbf{I} \quad \mathbf{YZ})$ is a well-defined power series. So eq. (89) can be viewed both as an equality between matrices of rational functions, and as an equality between matrices of power series.

For convenience, write $Q = \det(\mathbf{I} \quad \mathbf{YZ})$.

Since S and T are sets of size r, by linearity of the determinant we have

$$\det(\mathbf{I} \quad \mathbf{YZ})^{-1}[S;T] = \frac{\det(\text{adj}(\mathbf{I} \quad \mathbf{YZ}))[S;T]}{Q^r}. \tag{90}$$

By the previous discussion, the left hand side of eq. (90) is nonzero. Consequently, the numerator

$$\det(\text{adj}(\mathbf{I} \quad \mathbf{YZ}))[S;T]$$

on the right-hand side of eq. (90) must be a nonzero polynomial.

All entries of $\mathbf{Y}$ and $\mathbf{Z}$ have degree at most 1, so each entry of $\mathbf{YZ}$ has degree at most 2. Consequently, each entry of $\text{adj}(\mathbf{I} \quad \mathbf{YZ})$ has degree less than $2m_{\text{new}}$, which implies that the numerator polynomial from the above equation has total degree less than $2rm_{\text{new}}$.

Similarly, we observed earlier that Q is a polynomial with constant term 1, and is thus a nonzero polynomial. Hence the denominator

$$Q^r = (\det(\mathbf{I} \quad \mathbf{YZ}))^r$$

of the right hand side of eq. (90) is also a nonzero polynomial. As noted previously, every entry of $\mathbf{YZ}$ has degree at most 2. Thus, Q^r has degree at most $2rm_{\text{new}}$.

So the right hand side of eq. (90) is the ratio of two nonzero polynomials, each with degree at most $2rm_{\text{new}} \quad 2k(m + 2kn)$. Then by Corollary 7.7 and the choice of q in eq. (79), the random evaluation $\det(\mathbf{I} \quad \mathbf{YZ})^{-1}[S;T]$ is nonzero over F with probability at least

$$1 \quad 4k(m + 2kn)=(2^q) \quad 1 \quad 1=n^3$$

as desired.

Next, we will replace the matrix $(\mathbf{I} \quad \mathbf{BC})^{-1}$ from Lemma 7.20 with a smaller random matrix which encodes the connectivities in G.

Let

$$\widetilde{\mathbf{B}} = \mathbf{B}[E_{\text{out}}; \ ] \quad \text{and} \quad \widetilde{\mathbf{C}} = \mathbf{C}[\ ; E_{\text{in}}]$$

be the matrices obtained from $\widetilde{\mathbf{Y}}$ and $\widetilde{\mathbf{Z}}$ respectively by the variable assignment in eq. (87). In particular, $\widetilde{\mathbf{B}}$ is a $kn \quad kn_{\text{new}}$ matrix, and $\widetilde{\mathbf{C}}$ is a $kn_{\text{new}} \quad kn$ matrix.

Define the $kn \quad kn$ matrix

$$\mathbf{M} = \widetilde{\mathbf{B}}\,(\mathbf{I} \quad \mathbf{CB})^{-1}\,\widetilde{\mathbf{C}}:$$

Note that the matrix $\mathbf{M}$ here is different from the matrix of the same name defined earlier in the Exact subsection of Section 7.2.

Lemma 7.21 (Small Connectivities via Rank). With high probability, we have

$$\text{rank } \mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)] = \min(k; \ (s; t))$$

for all original vertices $s; t \ 2 \ V$.

Proof. Fix $s; t \ 2 \ V$. Let $= \ _{\text{new}}(s; t)$. By Lemma 7.17, k. By Lemma 7.20 and the definition of connectivity, with probability at least $1 \quad 1=n^3$, is the largest integer such that there exist subsets $S \quad E_{\text{out}}(s)$ and $T \quad E_{\text{in}}(t)$ of size with

$$\det(\mathbf{I} \quad \mathbf{BC})^{-1}[S; T] \not\equiv 0:$$

By the definition of matrix rank, this means that

$$= \text{rank}\,(\mathbf{I} \quad \mathbf{BC})^{-1}[E_{\text{out}}(s); E_{\text{in}}(t)]: \tag{91}$$

Now, by Lemma 7.19, we know that

$$(\mathbf{I} \quad \mathbf{YZ})^{-1} = \mathbf{I} + \mathbf{Y}(\mathbf{I} \quad \mathbf{ZY})^{-1}\mathbf{Z}:$$

Since s and t are original vertices, $E_{\text{out}}(s) \setminus E_{\text{in}}(t) = ?$, because all outgoing edges of s go to s_{out}, and all incoming edges of t come from t_{in}.

This means that $\mathbf{I}[E_{\text{out}}(s); E_{\text{in}}(t)]$ is the all zeros matrix. So restricting the rows and columns of both sides of the above equation to $E_{\text{out}}(s)$ and $E_{\text{in}}(t)$ respectively yields

$$(\mathbf{I} \quad \mathbf{YZ})^{-1}[E_{\text{out}}(s); E_{\text{in}}(t)] = \left(\mathbf{Y}(\mathbf{I} \quad \mathbf{ZY})^{-1}\mathbf{Z}\right)[E_{\text{out}}(s); E_{\text{in}}(t)]:$$

Under the assignment eq. (87), the above equation simplifies to

$$(\mathbf{I} \quad \mathbf{BC})^{-1}[E_{\text{out}}(s); E_{\text{in}}(t)] = \mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)]:$$

Combining the above equation with eq. (91), we see that with probability at least $1 \quad 1=n^3$,

$$\text{rank}\,\mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)] = \ _{\text{new}}(s; t):$$

By Lemma 7.17, since $s; t \ 2 \ V$ we have $_{\text{new}}(s; t) = \min(k; \ (s; t))$, so

$$\text{rank}\,\mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)] = \min(k; \ (s; t)):$$

This holds for any fixed choice of $s; t \ 2 \ V$. Then by a union bound over the n^2 pairs of original vertices $(s; t)$, we get that the above equation holds for all $s; t \ 2 \ V$ with probability at least $1 \quad 1=n$, as desired.

Algorithm 6. The k-Bounded All-Pairs Connectivity Algorithm

Inputs: A directed graph G, and positive integer k.
Returns: The value of $\min(k; \ (s;t))$ for each pair of vertices $(s;t)$ in G.

1. Compute $\mathbf{M} = \widetilde{\mathbf{B}}(\mathbf{I} \quad \mathbf{CB})^{\ 1}\widetilde{\mathbf{C}}$.
2. For each pair $(s;t)$ of original vertices, return

$$\operatorname{rank} \mathbf{M}[E_{\text{out}}(s); E_{\text{in}}(t)]$$

as the value for $\min(k; \ (s;t))$.

We present our algorithm for k-APC in Algorithm 6. By Lemma 7.21, this algorithm correctly solves k-APC with high probability. To show that Algorithm 6 is efficient, the following lemma will be helpful.

Lemma 7.22. We can compute the matrix $\mathbf{CB}$ in $O(n^2k^{\ l})$ time.

Proof. The rows and columns of $\mathbf{B}$ and $\mathbf{C}$ are indexed by pairs in $V_{\text{new}} \quad [k]$.

For any $(u;i);(v;j) \ 2 \ V_{\text{new}} \quad [k]$, we have

$$\mathbf{CB}[(u;i);(v;j)] = \sum_{e2E_{\text{out}}(u) \setminus E_{\text{in}}(v)} c_{ie}b_{ej}: \tag{92}$$

By assumption, G has at most k parallel edges between each pair of nodes. Then by its construction, G_{new} also has at most k parallel edges between each pair of its nodes. So for all pairs $(u;v)$, we have $jE_{\text{out}}(u) \setminus E_{\text{in}}(v)j \quad k$, so that the sum in the right hand side of the above equation has at most k terms. Computing this sum separately for each of the pairs $(u;v)$ and $(i;j)$ yields a simple $O(n^2k^3)$ time algorithm for computing $\mathbf{CB}$. To compute $\mathbf{CB}$ faster, we find its entries by multiplying various submatrices of $\mathbf{C}$ and $\mathbf{B}$.

For each pair of nodes $(u;v)$, let $\mathbf{C}_{uv}$ be the restriction of $\mathbf{C}$ to the rows indexed by $fug \quad [k]$ and the columns indexed by $E_{\text{out}}(u) \setminus E_{\text{in}}(v)$. Note that $\mathbf{C}_{uv}$ has k rows and at most k columns. Similarly, let $\mathbf{B}_{uv}$ be the restriction of $\mathbf{B}$ to the rows indexed by $E_{\text{out}}(u) \setminus E_{\text{in}}(v)$ and columns indexed by $fvg \quad [k]$. By construction, $\mathbf{B}_{uv}$ has k columns and at most k rows.

By eq. (92), for each pair of nodes $(u;v)$ we have

$$\mathbf{CB}[(u;i);(v;j)] = \mathbf{C}_{uv}\mathbf{B}_{uv}[(u;i);(v;j)]$$

for all pairs of nodes $(u;v)$ and indices $(i;j) \ 2 \ [k]^2$. So, we can find all entries of $\mathbf{CB}$ just by computing the product $\mathbf{C}_{uv}\mathbf{B}_{uv}$ for all pairs of nodes $(u;v)$.

There are n^2_{new} pairs of nodes $(u;v)$ to consider. For each such pair of nodes $(u;v)$, the product $\mathbf{C}_{uv}\mathbf{B}_{uv}$ can be computed in $O(k^{\ l})$ time since the number of rows and columns of each $\mathbf{C}_{uv}$ and $\mathbf{B}_{uv}$ is at most k. So the overall time to compute $\mathbf{CB}$ is asymptotically at most $(n_{\text{new}})^2k^{\ l} \quad O(n^2k^{\ l})$ which proves the lemma.

We are now ready to prove that k-APC can be solved in $O((kn)^{!})$ time.

Proof of Theorem 7.2. By Lemma 7.21, Algorithm 6 correctly solves the k-APC problem. It remains to prove that we can implement Algorithm 6 to run in $O((kn)^{!})$ time.

By Lemma 7.22, we can compute $\mathbf{CB}$ in $O(n^2k^{!})$ time. Having computed this matrix, we can then compute $(\mathbf{I} \quad \mathbf{CB})^{\ 1}$ in $O((kn_{\text{new}})^{!}) \quad O((kn)^{!})$ time by Proposition 7.5.

Then we can calculate

$$\mathbf{M} = \widetilde{\mathbf{B}}(\mathbf{I} \quad \mathbf{CB})^{\ 1}\widetilde{\mathbf{C}}$$

in $O((kn)^{!})$ additional time, because $\widetilde{\mathbf{B}}$ is $kn \quad kn_{\text{new}}$, $(\mathbf{I} \quad \mathbf{CB})^{\ 1}$ is $kn_{\text{new}} \quad kn_{\text{new}}$, and $\widetilde{\mathbf{C}}$ is $kn_{\text{new}} \quad kn$, where $n_{\text{new}} = 3n$.

So step 1 of Algorithm 6 can be implemented to run in $O((kn)^{!})$ time.

Step 2 of Algorithm 6 involves computing ranks of n^2 separate $k \quad k$ matrices, since for each pair of original vertices $(s; t)$, we have $\deg_{\text{out}}(s) = \deg_{\text{in}}(t) = k$ in G_{new}. By Proposition 7.6, step 2 then takes $O(k^2n^{!})$ time.

So overall Algorithm 6 runs in $O((kn)^{!})$ time as claimed.

7.3 Vertex Connectivity

In this section, we present algorithms for the k-APVC and k-Vertex Connectivity problems. Our approach is essentially identical to the strategy employed for solving APC in Section 7.2, but instead of enumerating collections of edge-disjoint walks we work with *internally vertex-disjoint walks*, because vertex connectivities are defined in terms of vertex-disjoint paths.

Graph Assumptions (Vertex Connectivity Case)

Throughout Section 7.3, we assume that G is a simple graph. This is without loss of generality, since for any fixed pair of vertices $(s; t)$, parallel edges in G between from u to v do not affect the value of $(s; t)$ unless $u = s$ and $v = t$. In the case where $u = s$ and $v = t$, each additional parallel edge from s to t simply increases the value of $(s; t)$ by one. So if G did have parallel edges, we could compute $(s; t)$ by first computing the vertex connectivity from s to t in the modified graph G^{0}, obtained by taking G and deleting any extra parallel edges between its nodes. After computing the s to t connectivity in G', we could then increase this value by the number of parallel edges from s to t which were deleted to go from G to G^{0} to recover the value of $(s; t)$. We also assume that G does not have any self-loops. This too is without loss of generality, since self-loops do not affect the connectivity between distinct pairs of vertices.

Since G has no parallel edges, $(s; t) \quad n \quad 1$ for all $s; t \ 2 \ V$. So without loss of generality, we assume that $k \quad n \quad 1$ in this section.

Throughout this section, $\deg_{\text{out}}(s)$ denotes the number of vertices which s has edges to, and $\deg_{\text{in}}(t)$ denotes the number of vertices which have edges to t.

Field Size

We recall the preliminaries from the Finite Field Computation subsection of Section 6.1. In particular we work over a field $\mathsf{F} = \mathsf{F}_{2^q}$. Throughout Section 7.3, we set q to be the smallest

positive integer with

$$2^q \quad 2n^5 \tag{93}$$

Note that we have $q = \quad (\log n)$.

Enumerating Walks

To solve vertex-connectivity algorithms, we follow the strategy from the Section 7.2, but use vertex-centered rather than edge-centered enumeration. Almost all the constructions and proofs in this section are symmetric to the reasoning used in Section 7.2, so for intuition and motivation for the arguments employed below, we refer the reader to Section 7.2.

For every edge $(u; v) \ 2 \ E$, we introduce an indeterminate variable x_{uv}. We use these variables to enumerate families of walks in G, following the discussion from the Node-Based subsection of Section 6.2. In particular, each walk W is assigned a monomial (W) recording the consecutive pairs of nodes it traverses as in eq. (56), and each collection $C = \hbar W_1; : : : ; W_r i$ of walks is assigned a monomial (C) according to eq. (57). Let $\mathbf{X}$ be the $n \quad n$ matrix with rows and columns indexed by V such that for each pair of vertices $(u; v)$ we have

$$\mathbf{X}[u; v] = \begin{cases} x_{uv} & \text{if } (u; v) \ 2 \ E \\ 0 & \text{otherwise:} \end{cases} \tag{94}$$

Note that this differs from the matrix of the same name from Section 7.2. Throughout this section, we will introduce matrices and other objects with the same names as, but different definitions from, those employed in Section 7.2.

Given vertices $u; v \ 2 \ V$ and an integer $` \quad 1$, let $W_`(u; v)$ be the set of all $u \quad v$ walks on exactly $`$ nodes. We can interpret eq. (94) as saying that the $(u; v)$ entry of $\mathbf{X}$ enumerates all walks on two nodes from u to v in G. These are precisely the walks in $W_2(u; v)$. The next result observes that higher powers of $\mathbf{X}$ enumerate longer walks in G.

Proposition 7.23. For any edges $e; f \ 2 \ E$ and integer $` \quad 0$, we have

$$\mathbf{X}^`[u; v] = \sum_{W \ 2 \ W_{`+1}(u;v)} (W):$$

Proof. By expanding out the definition of matrix multiplication, we see that

$$\mathbf{X}^`[u; v] = \sum_{\substack{w_1; : : : ; w_{`+1} \ 2 V \\ w_1 = u \\ w_{`+1} = v}} \prod_{j=1}^{` \quad 1} \mathbf{X}[w_j; w_{j+1}]:$$

By definition, $\mathbf{X}[w_j; w_{j+1}] = x_{w_j w_{j+1}}$ if $(w_j; w_{j+1}) \ 2 \ E$, and is zero otherwise. Thus, the product

$$\prod_{j=1}^{` \quad 1} \mathbf{X}[w_j; w_{j+1}]$$

is nonzero if and only if $W = \langle w_1, \ldots, w_\ell \rangle$ is a walk on ℓ nodes in G. In this case,

$$\prod_{j=1}^{\ell-1} \mathbf{X}[w_j, w_{j+1}] = \prod_{j=1}^{\ell-1} x_{w_j w_{j+1}} = \text{(W)}$$

so we have

$$\mathbf{X}^\ell[u, v] = \sum_{W \in W^{\ell+1}(u,v)} \text{(W)}$$

as claimed.

Corollary 7.24 (Enumerating Walks by Nodes). For any vertices $u, v \in V$, we have

$$(\mathbf{I} \quad \mathbf{X})^{-1}[u, v] = \sum_{\ell=0}^{n-1} \left(\sum_{W \in W^{\ell+1}(u,v)} \text{(W)} \right).$$

Proof. By eq. (69), every entry of $\mathbf{X}$ has constant term zero. So the claim follows by combining the geometric series formula from Proposition 6.3 with Proposition 7.23.

Enumerating Vertex-Disjoint Walks

Given subsets of vertices $S, T \subseteq V$ of equal size $|S| = |T| = r \geq 1$ and an integer $\ell \geq 1$, we define $F^\ell(S, T)$ to be the family of collections of r walks whose total size is ℓ, beginning at different nodes of S and ending at different nodes of T. If we fix some ordering $u_1, \ldots, u_r$ of the nodes in S, then we can view each element of $F^\ell(S, T)$ as a sequence of walks $\langle W_1, \ldots, W_r \rangle$ satisfying the properties that each W_i begins at u_i and ends at some node of T, the W_i walks all end at distinct nodes of T, and the sum of the sizes of the W_i walks is ℓ. We also define $D^\ell(S, T) \subseteq F^\ell(S, T)$ to be the family of collections of r *vertex-disjoint* walks from S to T of total size ℓ.

We will design enumerating polynomials for the $D^\ell(S, T)$ families using determinants. See the discussion before Idea 8 for intuition on why determinants are useful for this task.

Define $\mathbf{\Gamma} = (\mathbf{I} \quad \mathbf{X})^{-1}$. By Corollary 7.24, entries of $\mathbf{\Gamma}$ enumerate walks in G. We use this claim to prove that determinants of submatrices of $\mathbf{\Gamma}$ enumerate collections of walks in G.

Lemma 7.25 (Walks by Nodes). For equal-size subsets of nodes $S, T \subseteq V$, we have

$$\det \mathbf{\Gamma}[S, T] = \sum_{\ell=1}^{n} \left(\sum_{C \in F^\ell(S,T)} \text{(C)} \right)$$

Proof. Let $\mathsf{S}(S, T)$ be the set of all bijections from S to T.

By the definition of the determinant, we have

$$\det \mathbf{\Gamma}[S, T] = \sum_{\in \mathsf{S}(S,T)} \prod_{u \in S} \mathbf{\Gamma}[u, \quad (u)]. \tag{95}$$

Note that we do not include a factor for the sign of in the above equation, because we work over a field F of characteristic two.

By Corollary 7.24, for each $u \in S$ we have

$$\Gamma[u; \ \ (u)] = \sum_{\ell=0}^{1} \left(\sum_{W \in W_{\ell+1}(u; \ \ (u))} (W) \right) : \tag{96}$$

Write $S = \{u_1; \ldots; u_r\}$, where $r = |S| = |T|$.

By multiplying eq. (96) over all choices of $u \in S$, we have

$$\prod_{u \in S} \Gamma[u; \ \ (u)] = \prod_{i=1}^{r} \left(\sum_{\ell=0}^{1} \left(\sum_{W \in W_{\ell+1}(u_i; \ \ (u_i))} (W) \right) \right) : \tag{97}$$

Now, let L be the set of all r-tuples $(\ell_1; \ldots; \ell_r)$ of positive integers summing to

$$\ell_1 + \quad + \ell_r = \ell:$$

If we expand out the product on the right-hand side of eq. (97) and group terms according to the total length of the walks they come from, we obtain

$$\prod_{i=1}^{r} \left(\sum_{\ell=0}^{1} \left(\sum_{W \in W_{\ell+1}(u_i; \ \ (u_i))} (W) \right) \right) = \sum_{\ell=1}^{1} \left(\sum_{\substack{(\ell_1; \ldots; \ell_r) \in L \\ W_i \in W_{\ell_i}(u_i; \ \ (u_i))}} \prod_{i=1}^{r} (W_i) \right) :$$

The expression on the right-hand side above arises from distributing the product on the left-hand side of the equation over the sum of walks of all possible lengths from u_i to (u_i) for $i \in [r]$, and considering the lengths ℓ_i this walk could have.

By chaining the above equation together with eqs. (95) to (97), and interchanging summation, we get that

$$\det \Gamma[S; T] = \sum_{\ell=1}^{1} \left(\sum_{\in S(S;T)} \sum_{\substack{(\ell_1; \ldots; \ell_r) \in L \\ W_i \in W_{\ell_i}(e_i; \ \ (e_i))}} \prod_{i=1}^{r} (W_i) \right) : \tag{98}$$

To simplify eq. (98), observe that for any choice of bijection $\in S(S;T)$, lengths $(\ell_1; \ldots; \ell_r) \in L$, and walks $W_i \in W_{\ell_i}(u_i; \ \ (u_i))$, the collection $\langle W_1; \ldots; W_r \rangle$ is a sequence of walks from S to T of total size ℓ. Conversely, any collection $C \in F_\ell(S;T)$ has walks whose sizes sum up to ℓ, and corresponds to a unique bijection $\in S(S;T)$, obtained by checking which starting nodes in S are connected to which ending nodes in T by walks in C.

Thus, the inner nested summation above is equivalent to a single sum over all collections of walks in $F_\ell(S;T)$. Since the weight of a collection $C = \langle W_1; \ldots; W_r \rangle$ is

$$(C) = \prod_{i=1}^{r} (W_i);$$

the discussion from the previous paragraph together with Equation (98) implies that

$$\det \Gamma[S;T] = \sum_{\ell=1}^{n} \left(\sum_{C \in F_\ell(S;T)} \operatorname{sgn}(C) \right)$$

which proves the desired result.

Lemma 7.26 (Intersecting Walks Vanish). For any equal-size subsets of nodes $S, T \subseteq V$ and integer $\ell \geq 1$, we have

$$\sum_{C \in F_\ell(S;T)} \operatorname{sgn}(C) = \sum_{C \in D_\ell(S;T)} \operatorname{sgn}(C).$$

Proof. Fix $S, T \subseteq V$ and integer $\ell \geq 1$. Let $r = |S| = |T|$.

For convenience, abbreviate $F = F_\ell(S;T)$ and $D = D_\ell(S;T)$. Let $S = F \setminus D$ be the family of all collections of r walks beginning at different nodes of S and ending at different nodes of T, such that at least two walks in the collection intersect.

By the definition of S we have

$$\sum_{C \in F} \operatorname{sgn}(C) = \sum_{C \in D} \operatorname{sgn}(C) + \sum_{C \in S} \operatorname{sgn}(C).$$

So to prove the claim, it suffices to show that

$$\sum_{C \in S} \operatorname{sgn}(C)$$

is the zero polynomial. We prove this by pairing up collections C in S of equal weight $\operatorname{sgn}(C)$, and observing that contributions from these collections vanish modulo two.

Fix an ordering $u_1, \ldots, u_r$ of the nodes in S. Take any $C = \langle W_1, \ldots, W_r \rangle \in S$, with the walks ordered so that W_i begins at u_i. By assumption, at least two walks in C intersect. Let $i \in [r]$ be the smallest index such that W_i intersects some other walk in C. Let u be the first vertex on W_i which appears in another walk in C. Let $j \in [r]$ be the smallest index $j > i$ such that W_j passes through node u.

We can split the walk W_i uniquely

$$W_i = A_i \quad B_i$$

as the concatenation of a prefix A_i ending at u, and a suffix B_i which begins at u. We can similarly split W_j uniquely

$$W_j = A_j \quad B_j$$

as the concatenation of a prefix A_j ending at u, and a suffix B_j starting at u.

Now, define walks

$$W_i' = A_i \quad B_j \quad \text{and} \quad W_j' = A_j \quad B_i$$

by swapping the suffixes of W_i and W_j. Define a new collection C' of walks by replacing W_i and W_j in C with W_i' and W_j' respectively.

Since W_i and W_j end at different nodes of T, we know that $W_i' \not\equiv W_i$ and $W_j' \not\equiv W_j$. This shows that $C' \not\equiv C$. Since the walks in C' begin at different nodes of S and end at different nodes of T, $C' \in F$. Moreover, since W_i' and W_j' intersect, we have $C' \notin D$. Thus $C' \in S$.

Additionally, we claim that if we apply the above suffix swapping procedure (which we used to go from C to C') to the collection C', we recover C.

Indeed, for all $i \in [r]$ with $i < i$, the walk $W_i' = W_i$ does not intersect any other walk in C, by the definition of i. Since the set of vertices traversed by walks in $C \setminus \{W_i\}$ and $C \setminus \{W_i'\}$ are the same, this means that W_i' does not intersect any other walk in C' either. So i is also the smallest index in $[r]$ such that W_i' intersects some other walk in C. Since the u_i u prefixes of W_i' and W_i are the same, we see that u is also the first node in W_i' which appears in another walk of C'. Then because W_j' contains u, and $W_i' = W_i$ for all $i \neq i, j$, we see that $j > i$ is the smallest index such that W_j' contains u. Then when we swap the suffixes of W_i' and W_j' after the first appearance of u on these walks, we recover W_i and W_j respectively, and so applying the suffix swapping procedure to C' produces the original collection C as claimed.

So, the suffix swapping routine described above partitions S into distinct pairs.

Suppose C and C' are paired up by the suffix swapping argument. Then C and C' traverse the same multiset of consecutive pairs of nodes. Thus these collections

$$(C) = (C')$$

have the same weight. Since we work over a field of characteristic two, the above equation implies that each pair (C, C') of collections mapped to each other by suffix swapping satisfies

$$(C) + (C') = 0.$$

Since S is partitioned into such pairs, we have

$$\sum_{C \in S} (C) = 0.$$

Together with the discussion from the beginning of the proof, this proves the claim.

Corollary 7.27 (Vertex-Disjoint Walks)**.** For equal-size subsets of nodes S, T E, we have

$$\det \Gamma[S, T] = \sum_{\ell=1}^{1} \left(\sum_{C \in D_\ell(S,T)} (C) \right).$$

Proof. This follows by combining Lemmas 7.25 and 7.26.

Corollary 7.27 shows that determinants of submatrices of Γ produce enumerating polynomials for families consisting of collections of vertex-disjoint walks. We want to use these polynomials to compute vertex connectivity, which is defined in terms of vertex-disjoint *paths*, not walks. To next result shows that it is fine to work with walks instead of paths.

Lemma 7.28 (Vertex-Disjoint Walks) Vertex-Disjoint Paths)**.** Let S, T V be subsets of nodes of size $|S| = |T| = r$. If the graph G contains r node-disjoint walks from S to T, then G also contains r node-disjoint paths from S to T.

Proof. Let $\langle W_1, \ldots, W_r \rangle$ be a collection of vertex-disjoint walks from S to T in G. For each index $i \in [r]$, suppose W_i is a $u_i \to v_i$ walk, where $S = \{u_1, \ldots, u_r\}$ and $T = \{v_1, \ldots, v_r\}$.

For each $i \in [r]$, let G_i be the subgraph of G induced on the vertices by W_i. Let P_i be a shortest $u_i \to v_i$ path in G_i. These paths are vertex-disjoint, since they live in subgraphs on disjoint sets of vertices.

Thus $\langle P_1, \ldots, P_r \rangle$ is a collection of r vertex-disjoint paths from S to T in G, as desired.

Corollary 7.29. Let $S, T \subseteq V$ be subsets of nodes of size $|S| = |T| = r$. Then $\det \boldsymbol{\Gamma}[S, T]$ is a nonzero formal power series if and only if G contains r vertex-disjoint paths from S to T.

Proof. Suppose $C = \langle P_1, \ldots, P_r \rangle$ is a collection of r vertex-disjoint paths from S to T in G. Then the term $\mathbf{x}(C)$ occurs in the expansion of

$$\det \boldsymbol{\Gamma}[S, T] \tag{99}$$

given by Corollary 7.27. Moreover, any collection of paths $C' \neq C$ from S to T has weight $\mathbf{x}(C') \neq \mathbf{x}(C)$, because C consists of vertex-disjoint paths (so looking at the variables appearing in $\mathbf{x}(C)$, we can recover C uniquely). Hence, no other term from the summation in Corollary 7.27 produces the same monomial $\mathbf{x}(C)$. So $\mathbf{x}(C)$ appears in eq. (99) with nonzero coefficient, which implies that the determinant from eq. (99) is a nonzero power series.

Suppose now that G does not contain r vertex-disjoint paths from S to T. The contrapositive of Lemma 7.28 implies that G does not contain r edge-disjoint walks from S to T. Then Corollary 7.27 implies that eq. (99) is the zero polynomial. This proves the claim.

Random Evaluation

Having constructed polynomials to help enumerate collections of vertex-disjoint walks, we next want to evaluate these polynomials at random points, in order to efficiently detect large collections of vertex-disjoint walks (using the strategy discussed Chapter 6) and thus compute vertex connectivities.

We assign each edge $(u, v) \in E$ an independent, uniform random value a_{uv} in $\mathbb{F}$.

Let $\mathbf{A}$ be the matrix obtained from $\mathbf{X}$ by assing each variable x_{uv} the value a_{uv}. In other words, $\mathbf{A}$ is the random $n \times n$ edge-adjacency matrix of G defined by taking

$$\mathbf{A} = \begin{cases} a_{uv} & \text{if } (u, v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Let $\mathbf{M} = (\mathbf{I} - \mathbf{A})^{-1}$ be the evaluation of $\boldsymbol{\Gamma}$ under this same random assignment.

Lemma 7.30. Let $S, T \subseteq V$ be subsets of size $|S| = |T| = r$. If G contains r vertex-disjoint paths from S to T, then $\det \mathbf{M}[S, T]$ is nonzero with probability at least $1 - 1/n^3$. If instead G does not have r vertex-disjoint paths from S to T, then $\det \mathbf{M}[S, T]$ is zero.

Proof. By Corollary 7.29, the determinant

$$\det \boldsymbol{\Gamma}[S, T] \tag{100}$$

is a nonzero power series if and only if G contains r vertex-disjoint paths from S to T.

So if G does not contain r vertex-disjoint paths from S to T, then eq. (100) is the zero polynomial, so its random evaluation $\det \mathbf{M}[S;T]$ must equal zero as claimed.

Otherwise, G contains r vertex-disjoint paths from S to T, and eq. (100) is a nonzero power series. By the formula for the inverse of a matrix, we have

$$\mathbf{\Gamma}[S;T] = \frac{(\operatorname{adj}(\mathbf{I} \quad \mathbf{X}))\,[S;T]}{\det(\mathbf{I} \quad \mathbf{X})}. \tag{101}$$

The matrix $(\mathbf{I} \quad \mathbf{X})$ has ones across its main diagonal, and every other entry of this matrix has constant term zero Then by the formula for the determinant of a matrix, $\det(\mathbf{I} \quad \mathbf{X})$ is a polynomial with constant term 1, so by Proposition 6.2 the multiplicative inverse of $\det(\mathbf{I} \quad \mathbf{X})$ is a well-defined power series. As a consequence, eq. (101) can be viewed as an equality between matrices of formal power series.

For convenience, write $Q = \det(\mathbf{I} \quad \mathbf{X})$.

Since S and T are sets of size r, by linearity of the determinant we have

$$\det \mathbf{\Gamma}[S;T] = \frac{\det(\operatorname{adj}(\mathbf{I} \quad \mathbf{X}))[S;T]}{Q^r}. \tag{102}$$

By assumption, the left-hand side of eq. (77) is nonzero. Consequently, the numerator

$$\det(\operatorname{adj}(\mathbf{I} \quad \mathbf{X}))[S;T]$$

on the right-hand side of eq. (102) must be a nonzero polynomial. Since each entry of $\mathbf{X}$ has degree at most 1, each entry of $\operatorname{adj}(\mathbf{I} \quad \mathbf{X})$ has degree less than n, so the numerator polynomial from the above equation has total degree less than rn. Similarly, in the previous discussion we observed that Q is a polynomial with constant term 1, so the denominator

$$Q^r = (\det(\mathbf{I} \quad \mathbf{X}))^r$$

of the right hand side of eq. (102) has constant term 1, and is thus a nonzero polynomial as well. Since each entry of $\mathbf{X}$ has degree at most 1, this denominator has degree at most rn.

So the right hand side of eq. (102) is the ratio of two nonzero polynomials, each with degree at most $rn \quad n^2$. Then by Corollary 7.7 and eq. (93), the random evaluation $\det \mathbf{M}[S;T]$ is nonzero over F with probability at least

$$1 \quad 2n^2/(2^q) \quad 1 \quad 1/n^3$$

as desired.

We are now ready to establish a connection between vertex connectivity and ranks of submatrices of $\mathbf{M}$. We note that this connection is somewhat more complicated than the corresponding situation for connectivity in Lemma 7.16. This is because when $(s;t)$ is an edge, a maximum collection of internally vertex-disjoint $s \quad t$ paths may include the path $\langle s;t\rangle$ of length one, with no internal vertices. Such a situation does not occur when we work instead with edge-disjoint collections of paths. In the statement of the following lemma, recall that $V_{\text{out}}[s]$ and $V_{\text{in}}[t]$ denote the closed out-neighborhood of s and closed in-neighborhood of t, as defined in the discussion around eq. (67).

Lemma 7.31 (Vertex Connectivity as Rank). We have

$$\operatorname{rank} \mathbb{M}[V_{\text{out}}[s]; V_{\text{in}}[t]] = \begin{cases} (s;t) + 1 & \text{if } (s;t) \in E \\ (s;t) & \text{otherwise} \end{cases}$$

for all vertices $s; t \in V$, with probability at least $1 - 1/n$.

Proof. Fix vertices $s; t \in V$. Abbreviate $\lambda = \lambda(s;t)$. Let $r \geq 0$ be the largest integer such that there exist subsets $S \subseteq V_{\text{out}}[s]$ and $T \subseteq V_{\text{in}}[t]$ of size $|S| = |T| = r$ with the property that G contains r vertex-disjoint paths from S to T.

We will prove the lemma by bounding λ and r in terms of each other.

Claim 7.32. We have $r \leq \lambda$. If $(s;t) \in E$, we have the stronger bound $r \leq \lambda + 1$.

Proof. By definition, G contains λ distinct internally vertex-disjoint $s \to t$ paths $P_1; \ldots; P_\lambda$. The second vertex of each P_i must be a distinct node $u_i \in V_{\text{out}}(s)$. Similarly, the penultimate vertex of each P_i must be a distinct node $v_i \in V_{\text{in}}(t)$. Let

$$S = \{u_i \mid i \in [\lambda]\} \quad \text{and} \quad T = \{v_i \mid i \in [\lambda]\}.$$

As i ranges over $[\lambda]$, the $u_i \to v_i$ subpaths of P_i form a collection of vertex-disjoint paths from S to T. Thus $r \geq \lambda$ as claimed.

Now, suppose $(s;t) \in E$. Then we claim there exists $i \in [\lambda]$ such that $P_i = \langle s; t \rangle$.

Indeed, suppose to the contrary that no such i exists. Then $P_1; \ldots; P_\lambda; \langle s; t \rangle$ are a collection of more than λ internally vertex-disjoint $s \to t$ paths, which contradicts the definition of λ. So our initial assumption was false, and some P_i is equal to $\langle s; t \rangle$, as claimed.

Without loss of generality, suppose $P_1 = \langle s; t \rangle$. Then $u_1 = t$ and $v_1 = s$. In particular, we know that $u_i \neq t$ and $v_i \neq s$ for all $i \geq 2$.

Since S consists of the second nodes on $s \to t$ paths, we know that $s \notin S$. Similarly, since T consists of the penultimate nodes on $s \to t$ paths, we know that $t \notin T$.

Define sets

$$S' = S \cup \{s\} \quad \text{and} \quad T' = T \cup \{t\}.$$

By the previous discussion, $|S'| = |T'| = \lambda + 1$.

Moreover, the sequence $\langle s \rangle; \langle t \rangle; P_2; \ldots; P_\lambda$ consists of $(\lambda + 1)$ vertex-disjoint paths from the nodes in S' to the nodes in T'. Thus we have $r \geq \lambda + 1$ when $(s;t) \in E$, as desired. □

Claim 7.33. We have $r \geq \lambda + 1$. If $(s;t) \notin E$, we have the stronger bound $r \geq \lambda$.

Proof. By definition, there exist subsets $S \subseteq V_{\text{out}}[s]$ and $T \subseteq V_{\text{in}}[t]$ of size r such that the graph G contains r vertex-disjoint paths from S to T.

Let $P_1; \ldots; P_r$ be r vertex-disjoint paths from S to T of minimum total length.

Let u_i and v_i be vertices such that P_i is a $u_i \to v_i$ path for each $i \in [r]$.

For all $i \in [r]$, we claim that s cannot appear in P_i except as its first node, and t cannot appear in P_i except as its final node. Indeed, suppose to the contrary s is in P_i, yet $s \neq u_i$ is not the first vertex of P_i. If s is the final node of P_i, then $s \in T$. Since $s \neq t$, we then have $s \in V_{\text{in}}(t)$. Then replacing P_i with its single-node subpath $\langle s \rangle$ in the list $P_1; \ldots; P_r$ yields a collection of r vertex-disjoint paths from S to T of strictly smaller total length,

which contradicts the fact that the P_i were picked to minimize the total length among all such collections of disjoint paths. If $s \neq v_i$ is not the final node on P_i, let w be the vertex appearing immediately after s on P_i. By definition, $w \in V_{\text{out}}(s)$. Then replacing P_i with its subpath $P_i[w; v_i]$ in the list $P_1; \ldots; P_r$ yields a collection of r vertex-disjoint paths from S to T of strictly smaller total length, which again contradicts the minimality of the P_i. In either case we have a contradiction, so s cannot appear in P_i except as its first node, as claimed. Symmetric reasoning proves that t cannot appear in P_i, except as its final node.

For each $i \in [r]$, if $u_i \neq s$ define $A_i = \langle s; u_i i$. If instead $u_i = s$, set $A_i = \langle s i$.

Similarly, if $v_i \neq t$ define $B_i = \langle v_i; t i$, and if instead $v_i = t$, set $B_i = \langle t i$.

Now define the $s \quad t$ walks

$$Q_i = A_i \quad P_i \quad B_i:$$

Since s and t can only appear in a path P_i as its first node and final node respectively, the Q_i are simple $s \quad t$ paths. Moreover, since the P_i are vertex-disjoint paths, the Q_i are internally vertex-disjoint $s \quad t$ paths.

We analyze what happens if the Q_i paths are not all distinct.

Suppose that $Q_j = Q_l$ for some distinct $j; l \in [r]$.

We claim that in this case, P_j and P_l cannot both have length at least one. Suppose to the contrary that both P_j and P_l and have length at least one. We perform casework on the identities of the nodes u_j and u_l.

Suppose first that $s \in \{u_j; u_l\}$. Without of loss of generality, assume $u_j = s$, so that we have $A_j = \langle s i$. Now, if $v_j = t$, then $B_j = \langle t i$, so $P_j = Q_j = Q_l$ contains P_l as a subpath, which violates the assumption that P_j and P_l are vertex-disjoint. Hence $v_j \neq t$. Since t can only appear in P_j as its final node, this implies that P_j does not contain t. Let w be the second node in P_j. Since $u_j = s$ and P_j does not contain t, we know that $w \in \{s; t\}$. Since P_j and P_l are vertex-disjoint, P_l does not contain w. Then Q_l does not contain w either, because $w \in \{s; t\}$. Since Q_j contains w, this contradicts the assumption that $Q_j = Q_l$.

So suppose instead that $s \in \{u_j; u_l\}$. Then the second nodes of Q_j and Q_l are u_j and u_l respectively. Since P_j and P_l are vertex-disjoint, $u_j \neq u_l$. This implies that Q_j and Q_l have distinct second nodes, which again contradicts the assumption that $Q_j = Q_l$.

In either case we derive a contradiction, so P_j or P_l cannot both have length at least one, as claimed. So if $Q_j = Q_l$, at least one of P_j and P_l consists of a single node.

Without loss of generality, let P_j consist of a single node. Then u_j and v_j are both equal to the same vertex w, and this node satisfies $w \in V_{\text{out}}[s] \setminus V_{\text{in}}[t]$.

If $w \in \{s; t\}$, then $Q_j = \langle s; w; t i$. This means that $Q_l = \langle s; w; t i$ as well. Since $P_j = \langle w i$ and P_l are vertex-disjoint, it must be the case that $P_l = \langle s i$ or $P_l = \langle t i$. Either choice for P_l forces $Q_l = \langle s; t i$, which contradicts the assumption that $Q_j = Q_l$.

Thus $w \in \{s; t\}$ is forced. If $w = s$, then $P_j = \langle s i$ and $Q_j = \langle s; t i$. Then $Q_l = \langle s; t i$ as well. The only way P_l can be vertex-disjoint from P_j in this case is if $P_l = \langle t i$. Symmetric reasoning shows that if $w = t$, then we must have $P_l = \langle t i$ and $P_j = \langle s i$. Note that these two cases can only occur if $s; t \in V_{\text{out}}[s] \setminus V_{\text{in}}[t]$, which is equivalent to $(s; t) \in E$.

The above discussion shows that either all the Q_i paths are distinct, or exactly two of the Q_i are equal. Moreover, the latter case can only occur if $(s; t) \in E$.

Since at most two of the Q_i are equal, by removing at most one of the Q_i paths, we get a collection of $(r \quad 1)$ internally vertex-disjoint $s \quad t$ paths. The definition of vertex

connectivity then implies that $r \quad 1$, so $r \quad + 1$ as claimed.

If $(s; t) \notin E$, then the previous discussion shows that the Q_i are all distinct, and so form a collection of r internally vertex-disjoint $s \quad t$ paths, which forces $r \quad$ by the definition of vertex connectivity. □

By Claims 7.32 and 7.33, we get that $r = \quad + 1$ if $(s; t) \in E$, and $r = \quad$ if $(s; t) \notin E$. By the definition of r, there exist $S \quad V_{\text{out}}[s]$ and $T \quad V_{\text{in}}[t]$ of size r so that G contains r vertex-disjoint paths from S to T. By Lemma 7.30, we have

$$\det \mathbf{M}[S; T] \not\equiv 0$$

with probability at least $1 \quad 1=n^3$

Moreover, by Lemma 7.30 and the maximality of r, any $(r + 1) \quad (r + 1)$ submatrix of $\mathbf{M}[V_{\text{out}}[s]; V_{\text{in}}[t]]$ has determinant zero. Thus,

$$r = \operatorname{rank} \mathbf{M}[V_{\text{out}}[s]; V_{\text{in}}[t]]$$

with probability at least $1 \quad 1=n^3$.

Combined with the previous discussion relating r and , we get that

$$\operatorname{rank} \mathbf{M}[V_{\text{out}}[s]; V_{\text{in}}[t]] = \begin{cases} (s; t) + 1 & \text{if } (s; t) \in E \\ (s; t) & \text{otherwise} \end{cases}$$

with probability at least $1 \quad 1=n^3$.

This result holds for any fixed choice of $s; t \in V$. By a union bound over all n^2 choices of pairs of vertices, we get that the above equation holds for all $s; t \in V$ with probability at least $1 \quad 1=n$, as desired.

All-Pairs

Our goal in this subsection is to prove Theorem 7.3, by presenting an $O((kn)^{!})$ time algorithm for k-APVC. Given Lemma 7.31, a natural attempt at solving k-APVC is the following:

1. Compute $\mathbf{M} = (\mathbf{I} \quad \mathbf{A})^{\ 1}$.
2. Compute $\operatorname{rank} \mathbf{M}[V_{\text{out}}[s]; V_{\text{in}}[t]]$ for every $s; t \in V$, and use this information together with Lemma 7.31 to output the value of $\min(k; \ (s; t))$ for all $s; t \in V$.

Step 1 above takes $O(n^{!})$ time. Step 2 however takes at least $\deg_{\text{out}}(s) \deg_{\text{in}}(t)$ time to even read the $M[V_{\text{out}}[s]; V_{\text{in}}[t]]$ matrices it considers for each $s; t \in V$, and

$$\sum_{s;t \in V} \deg_{\text{out}}(s) \deg_{\text{in}}(t) = m^2$$

so this approach is too slow if we want to obtain a $O(k^2 n^{!})$ time algorithm for k-APVC.

Step 2 above is inefficient because G can contain vertices of high degree. Since we only need to compute vertex connectivity values less than or equal to k, one attempt to resolve

this issue (as suggested by Idea 9) is by modifying G in a way that reduces the outdegrees and indegrees of most vertices to be at most k, yet preserves k-bounded vertex connectivities.

This degree reduction approach was used by [AGI+18, Section 5]. They transform G by adding completely connected layers of k new vertices between a vertex and its out-neighbors and in-neighbors (a vertex-based analogue of the construction from the Reducing Degrees subsection of Section 7.2). Formally, they construct a new graph G_{new} with vertex set V_{new} by taking V, and introducing new vertices $u_{i;\text{out}}$ and $v_{i;\text{in}}$ for all $u; v \in V$ and $i \in [k]$. The edge set of G_{new} consists of edges from u to $u_{i;\text{out}}$ for all $i \in [k]$, edges from $v_{i;\text{in}}$ to v for all $i \in [k]$, and edges from $u_{i;\text{out}}$ to $v_{j;\text{in}}$ for all $(u; v) \in E$ and $(i; j) \in [k]$. Note that all $s; t \in V$ have outdegree and indegree k in G_{new}.

The argument in [AGI+18, Section 5] runs the proposed algorithm from steps 1 and 2 above on G_{new} to try and solve k-APVC. The transformation is useful in the sense that now in step 2 we only need to compute the ranks of $k \quad k$ matrices, so that this step can be implemented in $O(k^{!} n^2)$ time overall.

For our purposes, there are two issues with this degree reduction approach.

First, if $(s; t) \in E$, then the vertex connectivity from s to t in G_{new} is always k, independent of the value of $(s; t)$. So the above approach does not compute $\min(k; (s; t))$ for pairs of vertices $(s; t)$ which are edges in G.

Second, G_{new} has $O(kn)$ vertices, which means that although step 2 is greatly sped up, step 1 now runs in $O((kn)^{!})$ time, which is slower than the $O(k^2 n^{!})$ runtime we are aiming for in Theorem 7.3 if $! > 2$.

How can we resolve these issues, to prove Theorem 7.3?

The above transformation reduces outdegrees and indegrees to k by explicitly adding in layers of k new nodes, blocking the original vertices from their neighbors. Instead of adding in these layers explicitly, our goal will be to *simulate* them using matrix multiplications.

> **Idea 12** If we only need to detect up to k vertex-disjoint paths, then we can **simulate degree reduction** by taking k **random linear combinations** of the vectors corresponding to the incoming and outgoing paths in our enumeration.

Idea 12 turns out be a standard trick in the literature surrounding linear algebraic computations, and is useful in the design of fast algorithms for computing matrix rank (see e.g., the discussion in [CKL13, Section 1]). It can be viewed as another interpretation of Idea 11.

To implement the intuition of Idea 12, we define some new matrices.

For each pair $(i; u) \in [k + 1] \quad V$, we introduce an indeterminate y_{iu}.

Similarly, for each pair $(v; j) \in V \quad [k + 1]$, we introduce an indeterminate z_{vj}.

Let $\mathbf{Y}$ be the $(k + 1) \quad n$ matrix defined by setting

$$\mathbf{Y}[i; u] = y_{iu}$$

for every $i \in [k + 1]$ and $u \in V$.

Similarly, let $\mathbf{Z}$ be the $n \quad (k + 1)$ matrix defined by setting

$$\mathbf{Z}[v; j] = z_{vj}$$

for every $v \in V$ and $j \in [k + 1]$.

Ultimately, we will solve k-APVC by multiplying $\mathbf{M}$ by random evaluations of $\mathbf{Y}$ and $\mathbf{Z}$ to recover smaller matrices whose ranks help compute k-bounded vertex connectivities in G. To show correctness of this approach, we prove some lemmas characterizing the effect of multiplication by random matrices on rank.

Lemma 7.34 (Preconditioning). Let $\mathbf{H}$ be an $a \quad b$ matrix. Let $\mathbf{R}$ be a $(k+1) \quad a$ matrix with independent uniform random entries from F. Then

$$\operatorname{rank} \mathbf{RH} = \min(k+1; \operatorname{rank} \mathbf{H})$$

with probability at least $1 \quad (k+1)/|\mathsf{F}|$.

Proof. Since $\mathbf{RH}$ has $k+1$ rows, we know that $\operatorname{rank} \mathbf{RH} \quad k+1$. Since rank cannot increase under matrix products, we have $\operatorname{rank} \mathbf{RH} \quad \operatorname{rank} \mathbf{H}$. Thus

$$\operatorname{rank} \mathbf{RH} \quad \min(k+1; \operatorname{rank} \mathbf{H}): \tag{103}$$

To prove the claim, it remains to lower bound $\operatorname{rank} \mathbf{RH}$.

To that end, define $h = \min(k+1; \operatorname{rank} \mathbf{H})$.

By the definition of rank, there exist sets $S; T$ of size h such that $\mathbf{H}[S; T]$ is invertible.

Now, define the $(k+1) \quad a$ polynomial matrix $\widetilde{\mathbf{R}}$ by setting

$$\widetilde{\mathbf{R}}[i; j] = r_{ij}$$

where each r_{ij} is an indeterminate variable. Note that assigning each r_{ij} an independent, uniform random value from F would turn the polynomial matrix $\widetilde{\mathbf{R}}$ into $\mathbf{R}$.

Let I be an arbitrary subset of h rows of $\widetilde{\mathbf{R}}$.

B Claim 7.35. The determinant $\det \widetilde{\mathbf{R}}[I; S]\mathbf{H}[S; T]$ is a nonzero polynomial.

Proof. Consider the assignment of values from $\{0; 1\}$ to the r_{ij} variables which turns the matrix $\widetilde{\mathbf{R}}[I; S]$ into the identity matrix $\mathbf{I}$. Under this assignment, $\widetilde{\mathbf{R}}[I; S]\mathbf{H}[S; T]$ simplifies to $\mathbf{H}[S; T]$. This matrix is invertible, and thus has nonzero determinant. This shows that there is some assignment to the r_{ij} variables under which $\det \widetilde{\mathbf{R}}[I; S]\mathbf{H}[S; T]$ has nonzero evaluation, which implies that this determinant is a nonzero polynomial as claimed. □

Since each entry of $\widetilde{\mathbf{R}}$ has degree at most 1, the polynomial

$$\det \mathbf{R}[I; S]\mathbf{H}[S; T]$$

has degree at most h. By combining Claim 7.35 with Proposition 6.1, and using the fact that a random evaluation of the r_{ij} variables turns $\widetilde{\mathbf{R}}$ into $\mathbf{R}$, we get that

$$\det \mathbf{R}[I; S]\mathbf{H}[S; T] \not\equiv 0$$

with probability at least $1 \quad h/|\mathsf{F}|$. Since $h \quad k+1$, this probability is at least $1 \quad (k+1)/|\mathsf{F}|$.

By the definition of rank, this means that

$$\operatorname{rank} \mathbf{R}[\ ; S]\mathbf{H}[S; \ ] \quad r: \tag{104}$$

To relate the above rank to the rank of $\mathbf{RH}$, define the matrix $\mathbf{L}$ whose row set is the same as that of $\mathbf{H}$ (and thus also equal to the column set of $\mathbf{R}$) and whose columns are indexed by S, with the property that $\mathbf{L}[i; s] = 1$ if $i = s$, and $\mathbf{L}[i; s] = 0$ otherwise. Then by the definition of matrix multiplication, we have

$$\mathbf{R}[\ ; S]\mathbf{H}[S;\ ] = \mathbf{R}\mathbf{L}\mathbf{L}^{\top}\mathbf{H}:$$

Since rank cannot increase under matrix multiplication, by the above equation we have

$$\operatorname{rank}\mathbf{RH} \quad \operatorname{rank}\mathbf{R}\mathbf{L}\mathbf{L}^{\top}\mathbf{H} = \operatorname{rank}\mathbf{R}[\ ; S]\mathbf{H}[S;\ ] \quad r$$

where the last inequality follows from eq. (104).

Combining this inequality with eq. (103) proves the lemma.

Lemma 7.36 (Postconditioning). Let $\mathbf{H}$ be an $a \quad b$ matrix. Let $\mathbf{R}$ be a $b \quad (k+1)$ matrix with independent uniform random entries from F. Then

$$\operatorname{rank}\mathbf{HR} = \min(k+1; \operatorname{rank}\mathbf{H})$$

with probability at least $1 \quad (k+1)/|\mathsf{F}|$.

Proof. This follows from symmetric reasoning to the proof of Lemma 7.34.

Lemma 7.37 (Conditioning). Let $S; T \quad V$ be subsets of vertices. Then

$$\operatorname{rank}\mathbf{B}[\ ; S]\mathbf{M}[S; T]\mathbf{C}[T;\ ] = \min(k+1; \operatorname{rank}\mathbf{M}[S; T])$$

with probability at least $1 \quad 1/n^4$.

Proof. By Lemma 7.34 applied to $\mathbf{H} = \mathbf{M}[S; T]$ and $\mathbf{R} = \mathbf{B}[\ ; S]$, we get that

$$\operatorname{rank}\mathbf{B}[\ ; S]\mathbf{M}[S; T] = \min(k+1; \operatorname{rank}\mathbf{M}[S; T]) \tag{105}$$

with probability at least $1 \quad (k+1)/|\mathsf{F}|$. Assume eq. (105) holds.

Then by Lemma 7.36 applied to $\mathbf{H} = \mathbf{B}[\ ; S]\mathbf{M}[S; T]$ and $\mathbf{R} = \mathbf{C}[T;\ ]$ we have

$$\operatorname{rank}\mathbf{B}[\ ; S]\mathbf{M}[S; T]\mathbf{C}[T;\ ] = \min(k+1; \operatorname{rank}\mathbf{B}[\ ; S]\mathbf{M}[S; T])$$

with probability at least $1 \quad (k+1)/|\mathsf{F}|$. By eq. (105), the above equation implies that

$$\operatorname{rank}\mathbf{B}[\ ; S]\mathbf{M}[S; T]\mathbf{C}[T;\ ] = \min(k+1; \operatorname{rank}\mathbf{M}[S; T]):$$

By union bound over the applications of Lemmas 7.34 and 7.36, the above equation holds with probability at least $1 \quad (2k+2)/|\mathsf{F}|$. Since $k \quad n \quad 1$, we have $2k+2 \quad 2n$. By our choice of q in eq. (93), we see that $|\mathsf{F}| = 2^q \quad 2n^5$ so the above equation holds with probability at least $1 \quad 1/n^4$, as claimed.

For any pair of vertices $(s; t)$, we define the $(k+1) \quad (k+1)$ matrix

$$\mathbf{M}_{st} = \mathbf{B}[\ ; V_{\text{out}}[s]]\mathbf{M}[V_{\text{out}}[s]; V_{\text{in}}[t]]\mathbf{C}[V_{\text{in}}[t];\ ]: \tag{106}$$

The following result is the basis for the k-APVC algorithm.

Lemma 7.38. We have

$$\operatorname{rank} \mathbf{M}_{st} = \begin{cases} \min(k+1;\ (s;t)+1) & \text{if } (s;t) \in E \\ \min(k+1;\ (s;t)) & \text{otherwise} \end{cases}$$

for all vertices $s;t \in V$, with probability at least $1 - 2/n$.

Proof. Fix vertices s and t. By Lemma 7.37 applied to $S = V_{\text{out}}[s]$ and $T = V_{\text{in}}[t]$ we have

$$\operatorname{rank} \mathbf{M}_{st} = \min\left(k+1; \operatorname{rank} \mathbf{M}[V_{\text{out}}[s]; V_{\text{in}}[t]]\right) \tag{107}$$

with probability at least $1 - 1/n^4$.

So by union bound over all pairs of vertices $(s;t)$, eq. (107) holds for all $s;t \in V$ with probability at least $1 - 1/n^2$. Assuming eq. (107) holds, by Lemma 7.31, we have

$$\operatorname{rank} \mathbf{M}_{st} = \begin{cases} \min(k+1;\ (s;t)+1) & \text{if } (s;t) \in E \\ \min(k+1;\ (s;t)) & \text{otherwise} \end{cases}$$

for all vertices s and t, with probability at least $1 - 1/n$. By taking a union bound over this probability and the probability that eq. (107) hold for all $s;t \in V$, we see that the above equatio holds with probability at least $1 - 2/n$ as claimed.

By Lemma 7.38, we can compute k-bounded vertex connectivities by computing ranks of the $\mathbf{M}_{st}$ matrices. However, computing all the $\mathbf{M}_{st}$ matrices separately using the definition given in eq. (106) is too slow if we are aiming for an $O(k^2 n^{\omega})$ time algorithm. Instead, to compute the $\mathbf{M}_{st}$ efficiently, we will use the structure of G and the fact that the $\mathbf{M}_{st}$ matrices have common entries across different pairs of vertices $(s;t)$.

To perform this computation, we define some auxiliary matrices.

For each $i \in [k+1]$, let $\mathbf{P}_i$ be the diagonal matrix with rows and columns indexed by V, with $\mathbf{P}_i[u;u] = \mathbf{B}[i;u]$ for each $u \in V$. Similarly, for each $j \in [k+1]$, let $\mathbf{Q}_j$ be the diagonal matrix with rows and columns indexed by V, with $\mathbf{Q}_j[v;v] = \mathbf{C}[v;j]$ for each $v \in V$.

Also, let $\widetilde{\mathbf{A}}$ be the unweighted adjacency matrix of G. In other words, matrix $\widetilde{\mathbf{A}}$ is obtained by setting $x_{uv} = 1$ in $\mathbf{X}$ for every $(u;v) \in E$.

We can compute $\mathbf{M}_{st}$ using the above matrices with the following lemma.

Lemma 7.39. For any pair of vertices $(s;t)$ and indices $(i;j) \in [k+1]^2$, we have

$$\mathbf{M}_{st}[i;j] = \left(\widetilde{\mathbf{A}}\mathbf{P}_i\mathbf{M}\mathbf{Q}_j\widetilde{\mathbf{A}}^{>}\right)[s;t].$$

Proof. Since $\mathbf{P}_i$ and $\mathbf{Q}_j$ are diagonal matrices, the definition of matrix multiplication yields

$$\left(\widetilde{\mathbf{A}}\mathbf{P}_i\mathbf{M}\mathbf{Q}_j\widetilde{\mathbf{A}}^{>}\right)[s;t] = \sum_{u;v \in V} \widetilde{\mathbf{A}}[s;u]\mathbf{P}_i[u;u]\mathbf{M}[u;v]\mathbf{Q}_j[v;v]\widetilde{\mathbf{A}}^{>}[v;t]. \tag{108}$$

Substituting in the definitions of $\widetilde{\mathbf{A}}$, $\mathbf{P}_i$, and $\mathbf{Q}_j$, the right-hand side of eq. (108) equals

$$\sum_{\substack{u \in V_{\text{out}}(s) \\ v \in V_{\text{in}}(t)}} \mathbf{B}[i;u]\mathbf{M}[u;v]\mathbf{C}[v;j] = (\mathbf{B}[\,;V_{\text{out}}(s)]\mathbf{M}\mathbf{C}[V_{\text{in}}(t);\,])[i;j] = \mathbf{M}_{st}[i;j]$$

which proves the desired result.

Algorithm 7. The k-Bounded All-Pairs Vertex Connectivity Algorithm

Inputs: A directed graph G, and a positive integer k.
Returns: The value of $\min(k; \ (s;t))$ for each pair of vertices $(s;t)$ in G.

1. Compute $\mathbf{M} = (\mathbf{I} \quad \mathbf{A})^{\ 1}$.
2. For each pair $(i;j) \ 2 \ [k+1]^2$, compute the matrix
$$\mathbf{D}_{ij} = \widetilde{\mathbf{A}}\mathbf{P}_i\mathbf{M}\mathbf{Q}_j\widetilde{\mathbf{A}}^{>}:$$
3. For each pair of vertices $(s;t)$, compute the matrix $\mathbf{M}_{st}$ by setting
$$\mathbf{M}_{st}[i;j] = \mathbf{D}_{ij}[s;t]$$
for all $(i;j) \ 2 \ [k+1]^2$.
4. For each pair of vertices $(s;t)$, return
$$\begin{cases} (\operatorname{rank}\mathbf{M}_{st}) \quad 1 & \text{if } (s;t) \ 2 \ E \\ \min(k; \operatorname{rank}\mathbf{M}_{st}) & \text{otherwise} \end{cases}$$
as the value for $\min(k; \ (s;t))$.

We can now present our algorithm for solving k-APVC in Algorithm 7.

Lemma 7.40. Algorithm 7 solves k-APVC with high probability.

Proof. By Lemma 7.39, step 3 of Algorithm 7 correctly computes $\mathbf{M}_{st}$ for all $s;t \ 2 \ V$.
By Lemma 7.38, with high probability we have
$$\operatorname{rank}\mathbf{M}_{st} = \begin{cases} \min(k+1; \ (s;t)+1) & \text{if } (s;t) \ 2 \ E \\ \min(k+1; \ (s;t)) & \text{otherwise} \end{cases}$$
for all vertices s and t. Assume the above equation holds for all $s;t \ 2 \ V$.
Then if $(s;t) \ 2 \ E$, we have
$$(\operatorname{rank}\mathbf{M}_{st}) \quad 1 = \min(k+1; \ (s;t)+1) \quad 1 = \min(k; \ (s;t)):$$
If instead $(s;t) \ \not{2} \ E$, we have
$$\min(k; \operatorname{rank}\mathbf{M}_{st}) = \min(k; \min(k+1; \ (s;t))) = \min(k; \ (s;t))$$
So with high probability, Algorithm 7 returns the correct value of $\min(k; \ (s;t))$ in step 4 for every pair of vertices $(s;t)$, thus solving the k-APVC problem.

Proof of Theorem 7.3. By Lemma 7.40, Algorithm 7 solves k-APVC with high probability. To prove the theorem, it then suffices to show that Algorithm 7 can be implemented to run in $O(k^2 n^\omega)$ time.

Step 1 of Algorithm 7 takes $O(n^\omega)$ time by Proposition 7.5, since we compute $\mathbf{M}$ by inverting an $n \times n$ matrix.

Step 2 of Algorithm 7 computes, for each $(i, j) \in [k+1]^2$, the matrix $\mathbf{D}_{ij}$ by multiplying five $n \times n$ matrices. For each (i, j), this multiplication takes $O(n^\omega)$ time. Performing this computation for every pair $(i, j) \in [k+1]^2$ then takes $O(k^2 n^\omega)$ time overall.

Step 3 of Algorithm 7 involves going through the entries of all the $\mathbf{D}_{ij}$ matrices. This takes $O(k^2 n^2)$ time, since there are $(k+1)^2$ such matrices, each of which are $n \times n$.

Step 4 of Algorithm 7 involves computing the rank of $\mathbf{M}_{st}$ for all pairs of vertices (s, t). There are n^2 choices for (s, t), and each $\mathbf{M}_{st}$ is a $(k+1) \times (k+1)$ matrix. By Proposition 7.6, each individual rank computation takes $O(k^\omega)$ time. So, this step takes $O(k^\omega n^2)$ time overall. Since $k \leq n - 1$, this runtime is at most $O(k^2 n^\omega)$.

So Algorithm 7 runs in at most $O(k^2 n^\omega)$ time overall, as desired.

Global

In this subsection, we present our $O(n^\omega + nk^\omega)$ time algorithm for k-Vertex Connectivity, thereby proving Theorem 7.4.

Recall that in k-Vertex Connectivity, our goal is to determine if $\kappa(G) \geq k$.

If $\kappa(G) \geq k$, we say G is a *k-vertex connected* graph. In designing the k-Vertex Connectivity algorithm, it will be helpful to have a dual characterization of k vertex-connected graphs, in terms of "vertex-cuts" instead of vertex-disjoint paths. We introduce this definition next.

We say a graph is *disconnected* if it contains distinct nodes s, t such that no $s \rightsquigarrow t$ path exists in the graph. We say a subset of nodes C is a *vertex-cut* in G, if deleting the nodes in C from G produces a graph which is disconnected. If C is a vertex-cut of G, we also say that removing C *disconnects* G.

The following proposition asserts that k-vertex connected graphs are precisely those that cannot be disconnected by deleting fewer than k vertices.

Proposition 7.41 (Vertex-Cut Characterization). If G is not a complete graph, then for any fixed integer $k \leq n - 1$, we have $\kappa(G) \geq k$ if and only if every vertex-cut of G contains at least k vertices.

See [Fra11, Theorem 2.5.26] for a proof of Proposition 7.41.

We can of course solve k-Vertex Connectivity by solving the all-pairs problem k-APVC. However, this algorithm is too slow to establish Theorem 7.4. To solve k-Vertex Connectivity faster, we will employ Proposition 7.41

Suppose $\kappa(G) < k$. Since throughout Section 7.3 we assume that $k \leq n - 1$, this implies that G is not a complete graph. Then by Proposition 7.41, there exists a vertex-cut C of size less than than k. This means that for *any* vertex $w \notin C$, there exists a vertex v outside of C such that either every $w \rightsquigarrow v$ path passes through C, or every $v \rightsquigarrow w$ path passes through C. In particular, $\min(\kappa(w, v), \kappa(v, w)) < k$. This observation suggests the following strategy for solving k-Vertex Connectivity.

Idea 13 To check if $(G) < k$, instead of computing vertex connectivities for all pairs of nodes in G, it suffices to find a node w outside of a minimum-size vertex-cut of G, and then compute vertex connectivities to and from w to all other nodes. For small k, random sampling should find w outside of a minimum vertex-cut with good probability.

We also recall Menger's theorem, another dual characterization for maximums-size collections of vertex-disjoint paths in graphs.

Proposition 7.42 (Menger's Theorem: Vertex-Based). Let $S; T \quad V$ be subsets of vertices. The maximum size of a collection of vertex-disjoint paths beginning at S and ending at T is equal to the minimum number of vertices which must be deleted from G produce a graph which contains no S to T path.

See [Sch02, Theorem 9.1] for a proof of Proposition 7.42. We use Proposition 7.42 to deduce the following nice property of k-vertex connected graphs.

Proposition 7.43. If $(G) \quad k$, then for any subsets $S; T \quad V$ of size $jSj = jTj = k$, the graph G contains k vertex-disjoint paths from S to T.

Proof. We prove the contrapositive.

Suppose that $S; T \quad V$ are subsets of k vertices, such that G does not contain k vertex-disjoint paths from S to T. Then by Proposition 7.42, there exists a set C of fewer than k vertices, such that deleting C from G results in a graph with no S to T path.

Since $jCj < k = jSj = jTj$, we can pick vertices $s \; 2 \; S \, n \, C$ and $t \; 2 \; T \, n \, C$. By assumption, deleting C from G produces a graph with no $s \quad t$ path. So every $s \quad t$ path in G contains a vertex from C. This implies that

$$(s; t) \quad jCj < k$$

so $(G) < k$, which proves the desired result.

To use Idea 13 to solve k-Vertex Connectivity, we need an efficient way of checking if all the vertex connectivities to and from a given vertex w are at least k. We can do this using the random matrix $\mathbf{M}$ from the previous subsection. However, computing exact or even k-bounded connectivities to and from w using ranks of submatrices of $\mathbf{M}$ directly might be too slow if the vertex w has large indegree or outdegree. To get an efficient algorithm, we will make sure to only ever compute ranks of $k \quad k$ submatrices of $\mathbf{M}$. Intuitively, this will be fine for solving k-Vertex Connectivity because of the property of k-vertex connected graphs recorded in Proposition 7.43. The following subroutine forms the basis of the k-Vertex Connectivity algorithm.

Lemma 7.44. There is an algorithm Root_k, which given a vertex w and $\mathbf{M}$ returns YES or NO, and with probability at least $1 \quad 1=n^3$, has the following behavior:

1. if $\mathsf{Root}_k(w; \mathbf{M})$ returns YES, then $(s; w) \quad k$ and $(w; t) \quad k$ for all nodes $s; t \; 6= \; w$;
2. if $\mathsf{Root}_k(w; \mathbf{M})$ returns NO, then $(G) < k$.

Moreover Root_k runs in $O(n \quad (\min(k; n \quad k))^{!})$ time.

Proof. We first describe the steps of Root_k, and then prove it has the desired behavior.

We first scan through the graph and check that every vertex v in G has $\deg_{\text{in}}(v) \quad k$ and $\deg_{\text{out}}(v) \quad k$. If there is some vertex with indegree or outdegree less than k, we immediately return NO.

For the rest of the algorithm, we may then assume that the minimum indegree and outdegree in G is at least k.

We select a subset $S \quad V_{\text{out}}(w)$ of out-neighbors of w with $jSj = k$.

Similarly, we select a subset $T \quad V_{\text{in}}(w)$ of in-neighbors of w with $jTj = k$.

Now, for every vertex $s \in w$, we construct a set $S(s)$ as follows:

- if $s \notin T$, then we fix a subset $S(s) \quad V_{\text{out}}(s)$ with $jS(s)j = k$;
- if instead $s \in T$, then we fix a subset $S(s) \quad V_{\text{out}}[s]$ with $jS(s)j = k$ with the additional properties that that $s \in S(s)$ and $w \notin S(s)$.

Similarly, for every vertex $t \in w$, we construct a set $T(t)$ as follows:

- if $t \notin S$, then we fix a subset $T(t) \quad V_{\text{in}}(t)$ with $jT(t)j = k$;
- if instead $t \in S$, then we fix a subset $T(t) \quad V_{\text{in}}[t]$ with $jT(t)j = k$ with the additional properties that that $t \in T(t)$ and $w \notin T(t)$.

Having constructed these sets, we go through all vertices $s; t \in w$, and check if we have

$$\text{rank}\, M[S(s) \cap T; T \cap S(s)] \quad k \quad jS(s) \setminus Tj \tag{109}$$

and

$$\text{rank}\, M[S \cap T(t); T(t) \cap S] \quad k \quad jS \setminus T(t)j: \tag{110}$$

If eq. (109) holds for all $s \in w$ and eq. (110) holds for all $t \in w$, we return YES. Otherwise, if the above inequalities fail for any choice of $s; t \in w$, we return NO.

This completes the description of the algorithm. Next, we prove a series of claims that will help show correctness for the algorithm.

B Claim 7.45. Let $I; J \quad V$ be equal-size subsets of vertices. Let $\quad = jI \setminus Jj$. Then there are k vertex-disjoint paths from I to J in G if and only if there are $(k \quad)$ vertex-disjoint paths from $I \cap J$ to $J \cap I$ in G.

Proof. Suppose there is a collection of $(k \quad)$ vertex-disjoint paths from $I \cap J$ to $J \cap I$. Then for each vertex $v \in I \setminus V$, we can add the single-node path $\langle v \rangle$ to this collection, to get a collection of k vertex-disjoint paths from I to J.

Conversely, suppose there is a collection k vertex-disjoint paths from I to J. Delete any path from this collection which uses a node in $I \setminus V$. Since the paths are vertex-disjoint and $jI \setminus Vj$, the resulting collection has at least $(k \quad)$ paths. Moreover, each path in the collection cannot start at J or end at I, so we have a collection of $(k \quad)$ vertex-disjoint paths from $I \cap J$ to $J \cap I$.

This proves the claim. □

B Claim 7.46. The algorithm satisfies condition 1 from the statement of Lemma 7.44.

Proof. Suppose $\mathsf{Root}_k(w; \mathbf{M})$ returns YES. This means that eqs. (109) and (110) hold for all vertices $s; t \in w$. By applying Lemma 7.30 and Claim 7.45 to all pairs of sets of the form

$$(S(s) \cap T; T \cap S(s)) \quad \text{and} \quad (S \cap T(t); T(t) \cap S)$$

we deduce that there are k vertex-disjoint paths from $S(s)$ to T and from S to $T(t)$, for all vertices $s; t \in w$.

Take a vertex $s \in w$. Let $P_1; \ldots; P_k$ be k vertex-disjoint paths from $S(s)$ to T, where for each $i \in [k]$, P_i is a $u_i \quad v_i$ path. By definition of T, $v_i \in V_{\text{in}}(w)$. for all i.

If $s \notin T$, then $u_i \in V_{\text{out}}(s)$ for all i. Thus as i ranges over $[k]$, the paths

$$(s; u_i) \quad P_i \quad (v_i; w)$$

form a collection of k internally vertex-disjoint $s \quad w$ paths in G.

If instead $s \in T$, then $u_i \in V_{\text{out}}[s] \cap \{w\}$ for all i, and $u_i = s$ for some index i. Without loss of generality, let $u_k = s$. Since the P_i are vertex disjoint, $u_i \notin s$ for all $i < k$.

Then the length one path $\langle s; w\rangle$ together with

$$(s; u_i) \quad P_i \quad (v_i; w)$$

as i ranges over $[k \quad 1]$ form a collection of $(k \quad 1)$ internally vertex-disjoint $s \quad w$ paths.

Thus in either case, $(s; w) \quad k$ for all $s \in w$.

Symmetric reasoning shows that $(w; t) \quad k$ for all $t \in w$.

Thus condition 1 holds as claimed. □

B Claim 7.47. The algorithm satisfies condition 2 from the statement of Lemma 7.44.

Proof. Suppose $\mathsf{Root}_k(w; \mathbf{M})$ returns NO. There are two ways this could happen.

The first possibility is that the algorithm returns no because some vertex v has indegree or outdegree less than k. In this case $(G) < k$, since if $\deg_{\text{out}}(v) < k$, v cannot have k vertex-disjoint paths to any other vertex, and if $\deg_{\text{in}}(v) < k$, then v cannot have k vertex-disjoint paths coming in from any other vertex.

The second possibility is that eq. (109) does not hold for some $s \in w$, or eq. (110) does not hold for some $t \in w$.

If eq. (109) does not hold for a vertex $s \in w$, then it means that

$$\det \mathbf{M}[S(s) \cap T; T \cap S(s)] = 0:$$

By Lemma 7.30, with probability at least $1 \quad 1/n^3$, this means that G does not contain

$$k \quad |S(s) \setminus T|$$

vertex-disjoint paths from $S(s) \cap T$ to $T \cap S(s)$. By Claim 7.45, this means that G does not contain k vertex-disjoint paths to from $S(s)$ to T. Then by Proposition 7.43, $(G) < k$.

Symmetric reasoning shows that if eq. (110) does not hold for a vertex $t \in w$, that with probability at least $1 \quad 1/n^3$, we have $(G) < k$. □

B Claim 7.48. Given access to $\mathbf{M}$, the algorithm runs in $O(n \quad (\min(k; n \quad k))^{!})$ time.

Proof. Scanning through the graph to check that every vertex has indegree and outdegree at least k takes linear time. Constructing the sets S and T, as well as $S(s)$ and $T(t)$ for all $s; t \in w$, takes $O(m + kn)$ time, since we just need to scan through the neighborhoods of all vertices and select $2k$ nodes per vertex.

The only remaining step in the algorithm is to check whether eqs. (109) and (110) hold for all $s; t \in w$.

By construction, for all vertices $s; t \in w$ we have $|S(v)| = |T(v)| = |S| = |T| = k$.

Hence the sets $S(s) \cap T$ and $T(t) \cap S$ each have size at most k.

Since each $S(s)$ is a subset of vertices, we also know that

$$|S(s) \cap T| = |S(s)| \quad |S(s) \setminus T| \quad n \quad k$$

for all $s \in w$. Similar reasoning proves that $|T(t) \cap S| \quad n \quad k$ for all $t \in w$.

The above discussion shows that for all $s; t \in w$ the sets $S(s) \cap T$ and $T(t) \cap S$ have size at most $\min(k; n \quad k)$. Then by Proposition 7.6, for any fixed s, computing

$$\operatorname{rank}_{\mathbf{M}}[S(s) \cap T; T \cap S(s)]$$

takes $O((\min(k; n \quad k))^{!})$ time. Similarly, for any fixed t, computing

$$\operatorname{rank}_{\mathbf{M}}[T(t) \cap S; S \cap T(t)]$$

takes $O((\min(k; n \quad k))^{!})$ time.

The algorithm computes fewer than $2n$ such ranks.

Thus, the algorithm runs in at most $O(n \quad (\min(k; n \quad k))^{!})$ time, as claimed. □

By Claims 7.46 and 7.47 we get that Root_k has the claimed behavior. By Claim 7.48, Root_k runs in $O(n \quad (\min(k; n \quad k))^{!})$ time when given access to the entries of $\mathbf{M}$.

This proves the desired result.

Lemma 7.49. Algorithm 8 solves the k-Vertex Connectivity problem with high probability.

Proof. Suppose $(G) \quad k$. Then by the contrapositive of item 2 from the statement of Lemma 7.44, every call to Root_k in step 5 of Algorithm 8 returns YES with probability $1 \quad 1=n$. Hence the algorithm reaches step 6 and correctly returns YES in this case.

Otherwise, $(G) < k$. In this case, by Proposition 7.41, there is a subset $C \quad V$ on fewer than k vertices, such that deleting the nodes in C disconnects G.

Consider the nodes sampled in step 3 of Algorithm 8.

By our choice of $h = \lceil (\log n)=(\log(n=k)) \rceil$ in Algorithm 8, with probability at least

$$1 \quad (|C|=n)^h > 1 \quad (k=n)^h \quad 1 \quad 1=n$$

there is an index $j \in [h]$ such that the sampled node w_j is not in C.

Write $w = w_j$ for convenience. Let G^0 be the graph obtained by deleting all nodes of C in G. By definition, G^0 is disconnected.

Claim 7.50. There is a vertex $u \in C$ such that G^0 does not both have an $u \quad w$ path and a $w \quad u$ path.

Algorithm 8. The k-Vertex Connectivity Algorithm

Inputs: A directed graph G, and a positive integer k.
Returns: YES if $\kappa(G) \geq k$, NO if $\kappa(G) < k$.

1. Compute $\mathbf{M} = (\mathbf{I} - \mathbf{A})^{-1}$.
2. Set $h \leftarrow \lceil \log n / (\log(n/k)) \rceil$.
3. Sample nodes $w_1, \ldots, w_h$ independently and uniformly at random from V.
4. For each $i \in [h]$:
5. If $\mathsf{Root}_k(w_i, \mathbf{M})$ returns NO, return NO.
6. Return YES.

Proof. Suppose to the contrary that every vertex not in C contains paths to and from w in G'. Then for any vertices s and t in G^0, we can obtain an $s \rightsquigarrow t$ path in G^0 by concatenating shortest $s \rightsquigarrow w$ and $w \rightsquigarrow t$ paths in G^0. This contradicts the assumption that G^0 is disconnected, so our initial assumption was false and some node u satisfies the properties from the claim. □

Let $u \in C$ be a vertex satisfying the property from the statement of Claim 7.50. Without loss of generality, suppose that G^0 does not contain a $u \rightsquigarrow w$ path. Then every $u \rightsquigarrow w$ path in G uses a vertex in C. Consequently, $\kappa(u, w) \leq |C| < k$. By the contrapositive of item 1 of Lemma 7.44, we get that $\mathsf{Root}_k(w, \mathbf{M})$ returns NO, with probability at least $1 - 1/n^3$. So in step 5 of Algorithm 8, when $i = j$, Algorithm 8 correctly returns NO.

By a union bound over the h calls to Root_k, we see that Algorithm 8 solves k-Vertex Connectivity with high probability, as claimed.

Proof of Theorem 7.4. By Lemma 7.49, Algorithm 8 solves k-Vertex Connectivity with high probability. So to prove the theorem, it remains to show that Algorithm 8 can be implemented to run in $O(n^\omega + nk^\omega)$ time.

Step 1 of Algorithm 8 runs in $O(n^\omega)$ time by Proposition 7.5, since we compute $\mathbf{M}$ by inverting an $n \times n$ matrix.

Step 2 of Algorithm 8 takes constant time.

Step 3 of Algorithm 8 takes $h \cdot O(n)$ time.

In steps 4 and 5, we make h calls to Root_k using the matrix $\mathbf{M}$ we already computed. By Lemma 7.44, each call $O(n(\min(k, n - k))^\omega)$ time. To bound the total runtime of these steps, we consider two cases based off how large k is.

Case 1: $k \leq n/2$

Suppose that $k \leq n/2$. Then $\log(n/k) \geq 1$, so $h = \lceil \log n / (\log(n/k)) \rceil \leq \lceil \log n \rceil$. This means that the h calls to the Root_k subroutine take at most $O(nk^\omega)$ time overall.

Case 2: $k > n/2$

Suppose instead that $k > n/2$.

For any real $x \geq 0$, the Taylor series for the exponential function implies that

$$e^x \geq 1 + x.$$

By setting $x = (n - k)/k$ and taking logarithms of both sides above, we get that

$$\log(n/k) \leq ((n - k)/k).$$

This means that

$$h = \lceil (\log n)/(\log(n/k)) \rceil \leq O(k(\log n)/(n - k)).$$

Each call to the Root_k subroutine takes at most $O(n(n - k)^l)$ time. Then by the above bound, the h calls to the Root_k procedure take at most $O(nk(n - k)^{l-1})$ time overall.

Since $k > n/2$, we have $n - k < k$, and this time bound is at most $O(nk^l)$.

Thus in both case 1 and case 2, steps 4 and 5 of Algorithm 8 take $O(nk^l)$ time overall.

So Algorithm 8 takes $O(n^l + nk^l)$ time overall, as claimed.

7.4 Open Problems

Improved Algorithms

The algorithms we presented in this chapter use the algebraic framework of Chapter 6, and thus are randomized. It would be interesting to remove the need for randomness from these algorithms, while still preserving the runtime.

Open Problem 17. Can k-APC be solved in *deterministic* $O((kn)^l)$ time? Similarly, can k-APVC be solved in *deterministic* $O(k^2 n^l)$ time?

In Proposition 7.42 we recalled Menger's theorem, which related the existence of vertex-disjoint paths in a graph to the presence of vertex-cuts. A standard variant of this theorem provides an analogous characterization for edge-disjoint paths in graphs.

Proposition 7.51 (Menger's Theorem (Edge-Based))**.** Let s and t be vertices in G. Then the maximum number of edge-disjoint s t paths in G is equal to the minimum number of edges which must be deleted from G to produce a graph with no s t path.

See [Sch02, Corollary 9.1b] for a proof of Proposition 7.51.

Given nodes s and t, a set of edges C in G is called an $(s; t)$-*cut* if deleting all the edges in C from G results in a graph with no s t path. Proposition 7.51 gives a simple way of convincing someone that $(s; t)$ k: just present them with an $(s; t)$-cut of size at most k.

When solving k-APC, it would be nice if instead of just computing k-bounded connectivities for all pairs of nodes, we were also able to return small $(s; t)$-cuts that *certified* our algorithm returned the correct answer for $(s; t)$ in cases where this connectivity is reported to be less than k. Having access to such small $(s; t)$-cuts would allow for independent verification that the k-bounded connectivities reported by a k-APC algorithm are correct. This motivates the following question.

Open Problem 18 (Returning Small Edge-Cuts). Is there an $O((kn)^{!})$ time algorithm that, given a graph, returns for each pair of vertices $(s;t)$ with $(s;t) < k$ a set of $(s;t)$ edges whose removal produces a graph with no s t path?

Open Problems 17 and 18 have been resolved for constant k in the special case of directed acyclic graphs (DAGs) [AGI+19]. Specifically, over DAGs, there is a deterministic algorithm that returns an $(s;t)$-cut of minimum size for each pair of nodes $(s;t)$ in G with $(s;t)$ k, and runs in $(k \log n)^{4^k + o(k)} n^{!}$ time [AGI+18, Theorem 7.9]. The dependence on k in this runtime is quite high. Although for constant k the algorithm runs in $O(n^{!})$ time, already for k $(\log \log n)$ the algorithm does not run in polynomial time. Rather than directly resolving Open Problem 18, as a first step it may be easier to focus on the problem in DAGs, and see if we can improve the runtime dependence on k in the aforementioned algorithm for returning small $(s;t)$-cuts.

Open Problem 19. Given a directed acyclic graph and integer $k = (\log \log n)$, is there an $O(n^{!})$ time algorithm which returns, for each pair of vertices $(s;t)$ with $(s;t) < k$, an $(s;t)$-cut of size $(s;t)$? What about when $k = (\log n)$?

Theorem 7.2 shows that when k $O(1)$, k-APC can be solved in $O(n^{!})$ time. Can we achieve this runtime for some k which is polynomially related to n?

Open Problem 20. Does there exist a constant > 0 such that k-APC can be solved in $O(n^{!})$ time for $k = n$?

It would be very interesting to get algorithms which solve k-APC and k-APVC faster.

Open Problem 21. Can we solve k-APC in faster than $O((kn)^{!})$ time, and k-APVC in faster than $O(k^2 n^{!})$ time?

It would also be nice to obtain faster algorithms for k-APC and its variants in the special case of sparse graphs. Of course the $O(m^{!})$ time algorithm for APC already implies a result in this vein, but it still interesting to see if there are interesting ranges for the values of k and m in terms of n for which k-APC can be solved faster.

Open Problem 22. Can we solve k-APC faster in sparse graphs?

The above question can also be posed for the k-Vertex Connectivity problem. As discussed in Section 7.1, for any k 1 we can solve k-Vertex Connectivity in $n^{2+o(1)}$ time. This is almost-optimal in dense graphs, but we can hope for faster algorithms in sparse graphs. Indeed, we can solve k-Vertex Connectivity in $O(\min(mk^2; nk^3 + m^{1=2}nk^{3=2}))$ time [FNY+20, Theorem 5.2], which is truly subquadratic in n for sufficiently small m and k.

Open Problem 23. Can we solve k-Vertex Connectivity faster in sparse graphs?

One could also hope to show faster algorithms for the original APC problem.

Open Problem 24. Can we solve APC in faster than $\tilde{O}(\min(m^{!}; n^2 m^{1+o(1)}))$ time?

Many of the questions raised above can also be asked for the vertex-connectivity variant of APC. We did not formally define this problem in Section 7.1 (although we did discuss its relaxation, the k-APVC problem), so we introduce it now:

All-Pairs Vertex Connectivity (APVC)

Given a graph G, compute $(s; t)$ for all pairs of vertices $(s; t)$ in G.

Over general directed graphs, the current best algorithms for APVC have similar runtimes to the current fastest algorithms for APC. We mentioned in Section 7.1 that APC can be solved over *undirected* graphs in $\tilde{O}(n^2)$. In contrast, no near-quadratic algorithm is known for solving APVC in undirected graphs (in part because undirected graphs do not in general admit Gomory-Hu trees for vertex connectivity [Ben95]). We do know how to solve APVC over undirected graphs in $m^{2+o(1)}$ time however [Tra23, Theorem 1.2], which beats the $\tilde{O}(m^{!})$ time algorithm for APC in directed graphs if $! > 2$.

Open Problem 25. Is there a constant > 0 such that we can solve APVC on undirected graphs in $O(m^2)$ time?

If one believes that faster algorithms for APC should exist, then instead of trying to resolve Open Problem 24 directly, it may be easier to tackle the above question and try to design faster algorithms for APVC instead.

Better Lower Bounds

As mentioned in Section 7.1 (in the paragraphs after Theorem 7.1), there are popular hardness hypotheses in complexity theory that imply lower bounds on the time complexity for solving APC. These hypotheses posit the intractability of certain circuit analysis and graph theoretic problems.

Recall the kSAT problem, defined in the Reducing Width for SAT and #SAT subsection of Section 2.2. The current fastest algorithms for kSAT run in 2^{n} $^{(n=k)}$ time (see e.g., the discussion in [VW21, Section 1]). For superconstant k, these algorithms run in 2^{n} $^{o(n)}$ time. It is conjectured that this runtime is essentially optimal.

Definition 7.52 (SETH). The Strong Exponential Time Hypothesis (SETH) posits that for any $" > 0$, there exists an integer k 1 such that kSAT cannot be solved in $2^{(1}$ $^{")n}$ time.

SETH is a stronger assumption than P $\neq$ NP, and is applied extensively as a hardness hypothesis throughout computer science. Assuming SETH, we can use reductions from SAT to get lower bounds for the exact time complexity of computational problems of interest. It is known that refuting SETH requires making major progress in major open problems in circuit complexity [Wil13, JMV18], so that even if one believes SETH is false, reductions from SETH still imply *barriers* for obtaining faster algorithms for various problems.

Assuming SETH, APC requires at least $(mn)^{1-o(1)}$ time to solve [KT18, Theorem 1.8], and APVC requires at least $n^2 + m^{3=2-o(1)}$ time to solve [Tra23, Theorem 1.4]. Additional lower bounds for APC have been shown by assuming hardness of the following graph problem.

> 4-Clique
>
> Given an undirected graph G, determine if G has four distinct, mutually adjacent nodes.

The best known algorithms for 4-Clique rely on fast matrix multiplication. Given reals $a; b; c \geq 0$, we let $!(a; b; c)$ denote the rectangular matrix multiplication exponent, defined to be the smallest positive real such that we can compute the product of an $n^a \times n^b$ matrix and an $n^b \times n^c$ matrix in $n^{!(a;b;c)+o(1)}$ time. As with for the standard matrix multiplication exponent, we write $O(n^{!(a;b;c)})$ instead of $n^{!(a;b;c)+o(1)}$ for convenience.

The current fastest algorithms for 4-Clique run in $O(n^{!(1;2;1)})$ time [EG04]. The 4-Clique Hypothesis in fine-grained complexity posits that solving 4-Clique requires $n^{!(1;2;1)-o(1)}$ time to solve, i.e., the current algorithms for 4-Clique are essentially optimal. Although the 4-Clique Hypothesis is not as established as SETH, it has also been used as a source of conditional hardness in the literature, and identifies a key computational challenge that current methods in graph algorithms seem unable to overcome.

Assuming the 4-Clique Hypothesis, APVC requires at least $n^{!(1;2;1)-o(1)}$ time to solve [AGI+18, Section 4]. It seems that this reduction can be modified to show that APC requires at least $n^{!(1;2;1)-o(1)}$ time under the 4-Clique Hypothesis as well. More recently, it was shown that APVC over *undirected* graphs also requires $n^{!(1;2;1)-o(1)}$ time to solve under the 4-Clique Hypothesis [HLSW23]. If optimal matrix multiplication algorithms exist, then $!(1; 2; 1) = 3$. Using the current fastest rectangular matrix multiplication algorithms, we have $!(1; 2; 1) < 3:521$ [WXXZ24, Table 1].

So overall, the 4-Clique Hypothesis implies APC and APVC cannot be solved in truly subcubic time (which is the lower bound SETH implied for APC in *dense* directed graphs), and that solving these problems in $O(n^{3:5})$ time, for example, would require designing faster matrix multiplication algorithms.

Although these cubic lower bounds rule out the possibility of solving APC in directed graphs and APVC even in undirected graphs in near-quadratic time (under SETH or the 4-Clique Hypothesis), they remain far from the current best *quartic* time complexity upper bounds we have for these problems. Can we narrow this gap, and show better lower bounds for APC and APVC?

Open Problem 26. Can we prove, under some plausible hardness hypothesis, that APC requires subcubic time to solve, even if $!(1; 2; 1) = 3$?

As discussed above, the current best lower bounds for APC (under the 4-Clique Hypothesis) hold even for the seemingly simpler problem of solving APVC on undirected graphs. This is interesting, since we know how to solve the latter problem in $m^{2+o(1)}$ time, while no such algorithm is known for APC. Can we show better hardness results for APC? Or do APC and APVC actually have the same complexity?

Open Problem 27. Does some plausible hardness hypothesis imply a time lower bound for APC in directed graphs that does not also hold for APVC in undirected graphs?

Open Problem 28. Are there efficient reductions between APC and APVC which show the time complexities of these problems are essentially equivalent in directed graphs?

Conditional lower bounds have also been studied for the relaxations k-APC and k-APVC. Under SETH, solving k-APC requires $(kn^2)^{1\ o(1)}$ time [KT18, Theorem 4.3]. Assuming the 4-Clique Hypothesis, solving k-APVC requires $(k^2n^{!(1;2;1)\ 2})^{1\ o(1)}$ time [AGI+18, Lemma 4.4], and it seems that this reduction can be modified to show the same lower bound for k-APC.

Again, these lower bounds are very far from the current best time upper bounds we have for k-APC and k-APVC. In fact, if $!(1;2;1) = 3$, then the lower bounds under the 4-Clique Hypothesis are only nontrivial if $k \quad p\overline{n}$. Can we narrow this gap, by showing better conditional lower bounds for these problems?

Open Problem 29. Can we prove better conditional lower bounds for the time complexities of k-APC and k-APVC in terms of k and n? What about for the special case where $k \quad p\overline{n}$.

We can also ask the analogue of Open Problem 28 for k-APC and k-APVC.

Open Problem 30. Are there efficient reductions between k-APC and k-APVC which show the time complexities of these problems are essentially equivalent in directed graphs, at least for certain interesting ranges of the parameter k?

Faster Verification

In Section 2.1, we discussed *verifiers* in the context of the complexity class NP. Trying to design efficient verifiers for APC and its variants is a fascinating research direction, which seems closely tied to the problems discussed previously of designing faster algorithms and better conditional lower bounds for these tasks.

In this section, a verifier is an algorithm which takes as input both the problem instance and a *certificate*, and returns YES or NO. We view the certificate as a message which includes a claim for what the answer to the problem is, together with a succinct proof explaining why the answer is correct. We say the verifier is *deterministic* if its underlying algorithm is deterministic. A deterministic verifier correctly solves a problem provided it returns YES if

and only if the claimed answer in the certificate is the correct answer to the input problem instance. A verifier solves a problem instance I in time bound $T(I)$ if there exists a certificate c whose claim for the answer to I is correct, such that the verifier returns YES in time $T(I)$ when given I and c as input. In other words, the runtime of a verifier for a given problem instance is defined by taking the minimum runtime of the algorithm over all certificates which make the verifier solve that instance correctly. The overall runtime of the verifier is, as usual, is its worst-case runtime over all problem instances.

Finding efficient verifiers for problems is interesting for at least two reasons. First, the design of an efficient verifier for a task may influence and inform the design of faster classical algorithms for the problem. For example, researchers first designed $\tilde{O}(n^2)$ deterministic verifiers for APC on undirected graphs [AKT20], before eventually showing that APC could be solved outright in $\tilde{O}(n^2)$ time. Second, the runtimes of verifiers for a task can provide insight into hardness reductions for that problem, and thereby help inform our beliefs about the true time complexity of the tasks we study, and in some cases conditionally rule out the existence of certain hardness reductions [CGI+16].

The APVC problem admits a deterministic verifier running in $O(n^{!(1;2;1)})$ time [Tra23, Lemma 2.3]. This runtime is interesting, because it matches the fastest known runtime for solving 4-Clique, using a standard algorithm or deterministic verifier. Since the current best conditional lower bound for APVC comes via reduction from 4-Clique, this coincidence in runtimes naturally motivates the question of whether 4-Clique or APVC have faster deterministic verifiers. If the fastest deterministic verifiers for these problems turn out to have the same runtime, that would give circumstantial evidence that APVC and 4-Clique should have the same complexity.

Open Problem 31. Is there a constant $\quad > 0$, such that 4-Clique admits a deterministic verifier running in $O(n^{!(1;2;1)\quad})$ time?

Since existing 4-Clique algorithms rely on matrix multiplication, resolving Open Problem 31 may be connected to designing fast deterministic verifiers for multiplying matrices. This is an interesting open question in its own right, where not much progress has been made. See [Kün18] for some discussion of this problem.

Although APVC can be solved by a deterministic verifier running in $O(n^{!(1;2;1)})$ time, the current best deterministic verifier for APC takes $O(n^{!(1;2;1)} + n^{5=2}\overline{m})$ time [Tra23, Lemma 2.4], which is slower than the former runtime in dense graphs. Can we close the gap between verifier runtimes for APVC and APC?

Open Problem 32. Does APC have a deterministic verifier running in $O(n^{!(1;2;1)})$ time?

It would also be interesting to obtain nontrivial deterministic verifiers for the parameterized relaxations of APC and APVC.

Open Problem 33. Are there deterministic verifiers which solve k-APC in faster than $O((kn)^{!})$ time, or solve k-APVC in faster than $O(k^2n^{!})$ time?

Note that the current best algorithms for k-APC and k-APVC are randomized, so a priori it is not clear whether these problems even admit $O((kn)^{l})$ time verifiers.

We can also consider *randomized verifiers*, where the underlying algorithm of the verifier can use randomness, and we only require that the verifier correctly checks a claimed answer to a with high probability, for each fixed problem instance and certificate. It seems possible that known algorithms for APC and its variants can be modified to obtain randomized verifiers using standard techniques for verifying matrix products (e.g., see "Freivalds' Technique" in [MR95, Section 7.1]). The time complexity of randomized verifiers for these problems has not been studied closely in the literature, and is worth investigating further.

Open Problem 34 (Randomized Verification). How quickly can we solve APC and its variants using randomized verifiers?

Extending Techniques

Given a graph G with vertex set V, and distinguished subsets $S; J \quad V$, the *gammoid* on J with respect to S is the collection I of subsets $T \quad J$ with the property that G contains jTj vertex-disjoint paths beginning at nodes in S and ending at the nodes in T.

Gammoids have many applications in algorithm design [FLSZ18, Chapter 10]. These applications rely on the fact that there are randomized polynomial-time algorithms for constructing *linear representations* of gammoids. A linear representation of the gammoid on set J with respect to S is a matrix $\mathbf{M}$, whose columns are indexed by vertices in J, with the property that for any fixed $T \quad J$, with high probability we have $T \; 2 \; I$ if and only if $\mathbf{M}[\,; T]$ has full rank. Note that Lemma 7.30 gives an explicit construction for a randomized linear representation of a gammoid, and Lemma 7.15 does the same for an *edge-disjoint* analogue of the gammoid.

For most algorithmic applications in the literature, the existence of linear representations of gammoids can be used as a black-box. However, the enumerative perspective we used to prove Lemma 7.15 was important for obtaining results like Lemma 7.18, which were in turn crucial for designing and proving correctness of our k-APC algorithm. It is not immediately clear if Theorem 7.2 could be proved, for example, by using representations of gammoids in a black-box fashion. Could the enumerative perspective give speed-ups for other problems related to gammoids?

Open Problem 35. Can the the low-rank enumeration used Algorithm 6 to solve k-APC help solve other problems involving gammoids? Conversely, can existing algorithms which leverage computation on gammoids provide insights into the k-APC problem and its variants?

The Laplacian matrix $\mathbf{L}$ of a graph is an important object of study in fast algorithms for maximum flow. One of the key properties of $\mathbf{L}$ is that it can be factored into the product of two "edge-vertex incidence" matrices. The low-rank factorization $\mathbf{X} = \mathbf{YZ}$ employed in our k-APC algorithms feels similar to this factorization. Could there be some advantage to

combining Laplacian-based techniques (such as those discussed in [Vis13]) and the arguments used in our k-APC algorithm?

Open Problem 36. Can optimization techniques, such as approximate Laplacian solvers, help design faster algorithms for k-APC?

The low-rank factorization in used in our k-APC intuitively reduces the complexity of an *edge-disjoint* problem (computing connectivities) to something like a *vertex-disjoint* variant of the problem (for example, this is suggested by Figure 3). Can we make this connection formal, and use low-rank enumeration to solve other problems involving edge-disjoint paths?

Open Problem 37. Can the low-rank enumeration used in Algorithm 6 to solve k-APC help design faster algorithms for other problems involving edge-disjoint paths? Can it help design more efficient reductions from edge-disjoint to vertex-disjoint path problems?

Chapter 8

Disjoint Shortest Paths

8.1 Overview

In Section 7.1, we introduced the idea of using disjoint paths to measure connectivity in networks, and more generally discussed the significance of *routing disjoint paths* on graphs in mathematics and computer science. Finding connectivities involves detecting disjoint paths between a fixed pair of vertices s and t. A natural generalization of this task is to seek disjoint paths which connect *multiple terminal pairs* in the graph. Beyond being mathematically interesting, such tasks arise in when studying questions concerning transportation networks and circuit layout.

This motivates the following disjoint paths problem: for a fixed positive integer k, in the k-Disjoint Paths (k-DP) problem, we are given a graph G, with specified source nodes $s_1, \ldots, s_k$ and target nodes $t_1, \ldots, t_k$, and are tasked with determining if G contains internally vertex-disjoint s_i t_i paths. Note that the paths in a solution to k-DP are required to pair each source s_i to its corresponding target t_i. This condition turns out to make k-DP difficult in general—if we were tasked with looking for k disjoint paths connecting each source node to *any* target node instead of its corresponding target, then the problem would be easy to solve using a simple reduction to maximum flow.

Suppose the input graph G has n vertices and m edges. How quickly can we solve k-DP?

If k is allowed to be unbounded, the k-DP problem becomes NP-hard even on very simple classes of graphs [MP93], and over general directed graphs, k-DP is NP-hard already for $k = 2$ [FHW80, Lemma 3]. In light of these hardness results, for the purpose of designing polynomial-time algorithms for disjoint path problems, we should focus on the case where the input comes from a restricted class of graphs and k is constant. One such class, which has received extensive attention in the literature surrounding disjoint path problems, is the class of undirected graphs.

Undirected Graphs

Over undirected graphs, k-DP can be solved in $O(m + n)$ time for $k = 2$ [Tho05], and more generally for any constant k in $O(n^2)$ time [KKR12] and $m^{1+o(1)}$ time [KPS24]. The algorithms for k 3 are obtained using deep connections between the k-DP problem and the theory of forbidden minors in undirected graphs [RS95]. So over undirected graphs, the

polynomial-time complexity of k-DP is essentially resolved: for unbounded k the problem is NP-hard, and thus unlikely to be polynomial-time solvable, and for constant k the problem can be solved in almost-linear time.

In this chapter, we study an optimization variant of k-DP, the k-Disjoint Shortest Paths (k-DSP) problem. In k-DSP we are given the same input as in k-DP, but are now tasked with determining if the input contains disjoint s_i t_i *shortest* paths.

k-Disjoint Shortest Paths (k-DSP)

Given a graph G with specified nodes $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$, determine if G contains internally vertex-disjoint s_i t_i shortest paths.

The k-DSP problem is interesting both because it is a natural graph algorithms question to investigate from the perspective of combinatorial optimization, and because understanding the complexity of k-DSP could lead to a deeper understanding of the interaction between shortest paths structures in graphs (analogous to how studying k-DP helped develop the rich theory surrounding forbidden minors in graphs [RS95]).

Compared to k-DP, not much is known about the exact time complexity of k-DSP. In directed graphs, 2-DSP can be solved in polynomial time [BK17], but no polynomial-time algorithm (or NP-hardness proof) is known for k-DSP for *any* constant k 3. In undirected graphs, it is known that for any constant k, k-DSP can be solved in polynomial time [Loc21]. However, the current best algorithms for k-DSP in undirected graphs run in $n^{O(k\,k!)}$ time, so in general this polynomial runtime is quite large for k 3. For example, the current fastest algorithm for 3-DSP in undirected graphs takes $O(n^{292})$ time [BNRZ21].

Significantly faster algorithms are known for detecting $k = 2$ disjoint shortest paths. In this case, we allow G to be a weighted graph with positive edge weights (of course the k-DSP problem makes sense on weighted graphs for any k, but currently there are no algorithms published which explicitly handle the case of weighted graphs for any k 3). The paper which first introduced the k-DSP problem in 1998 also presented an $O(n^8)$ time algorithm for solving 2-DSP in weighted undirected graphs [ET98]. This algorithm was improved upon for the first time over twenty years later in [Akh20], which presented an algorithm solving 2-DSP in weighted undirected graphs in $O(n^7)$ time, and in unweighted undirected graphs in $O(n^6)$ time. Soon after, [BNRZ21, Theorem 1] presented an even faster $O(mn)$ time algorithm for solving 2-DSP in the special case of unweighted undirected graphs.

Despite all the advances discussed above, there remains a gap between the fastest runtimes known for solving the 2-DSP and 2-DP problems in undirected graphs. We close this gap, presenting an optimal algorithm for 2-DSP in weighted undirected graphs.

Theorem 8.1: 2-Disjoint Shortest Paths in Undirected Graphs

There is an algorithm solving 2-DSP on undirected graphs in linear time.

Directed Acyclic Graphs

Besides undirected graphs, another class of graphs for which research on disjoint path problems has been particularly fruitful is the class of directed acyclic graphs (DAGs). Designing

algorithms for detecting disjoint paths on DAGs appears to be especially important because such algorithms have been used as key subroutines for finding disjoint paths in other types of graphs. For example, the only known polynomial-time algorithm for 2-DSP on general directed graphs works by reducing to several instances of 2-DP on DAGs [BK17]. Similarly, the fastest known algorithm for k-DSP on undirected graphs works by reducing to several instances of disjoint paths on DAGs [BNRZ21].

Over DAGs, k-DP can be solved in linear time for $k = 2$ [Tho12], and in $O(mn^{k-1})$ time for $k \geq 3$ [FHW80, Theorem 3]. As observed in [BK17, Proposition 10], the algorithm of [FHW80] for k-DP on DAGs can be modified to solve k-DSP in weighted DAGs in the same $O(mn^{k-1})$ runtime. This is the fastest known runtime for k-DSP in DAGs for all k.

In particular, the fastest algorithm for 2-DSP from previous work runs in $O(mn)$ time, slower than the $O(m + n)$ runtime known for 2-DP in DAGs. As in the case of undirected graphs, we close this gap, presenting an optimal algorithm for 2-DSP in weighted DAGs.

Theorem 8.2: 2-Disjoint Shortest Paths in Directed Acyclic Graphs

There is an algorithm solving 2-DSP on directed acyclic graphs in linear time.

As we discuss in Section 8.6, Theorem 8.2 implies an alternate linear time algorithm for 2-DP in DAGs, which may be interesting due to its simplicity and the different techniques it employs compared to [Tho12].

Our algorithms for solving 2-DSP in undirected graphs and DAGs are algebraic, following the framework of Chapter 6. As a consequence, the algorithms establishing Theorems 8.1 and 8.2 determine whether the input graph contains disjoint $s_i \leadsto t_i$ shortest paths, but do not explicitly return these solution paths when they exist. So while our algorithms solve the decision problem 2-DSP, they do not solve the search problem of returning two disjoint shortest paths if they exist. This is a common limitation for algebraic graph algorithms. We show nonetheless that using a simple search to decision reduction for 2-DSP, we can bootstrap Theorems 8.1 and 8.2 to obtain $O(mn)$ time algorithms which *find* two disjoint shortest paths in weighted undirected graphs and DAGs, or report that no such paths exist.

Theorem 8.3: Finding 2 Disjoint Shortest Paths

There is an algorithm which solves 2-DSP over weighted DAGs and undirected graphs, and returns a pair of solution paths if they exist in $O(mn)$ time.

Organization

In Section 8.2, we go over useful notation and results on graphs. In Section 8.3, we introduce the basic ideas behind our 2-DSP algorithms, and prove lemmas which apply to 2-DSP on both DAGs and undirected graphs. In Section 8.4, we present our algorithm for 2-DSP in weighted DAGs, and prove Theorem 8.2. In Section 8.5, we present our algorithm for 2-DSP in weighted undirected graphs, and prove Theorem 8.1.

In Section 8.6, we observe some simple corollaries of Theorems 8.1 and 8.2. Finally, in Section 8.7, we conclude with relevant open problems.

8.2 Preliminaries

We make use of the preliminaries from Section 6.1. We introduce some additional graph theoretic notation, assumptions, and concepts below.

Basic Graph Notation and Assumptions

Throughout, we let G be the input graph on n vertices and m edges. We assume G is a simple graph (i.e., has no parallel edges between nodes). We let $s_1; s_2$ and $t_1; t_2$ denote the source nodes and target nodes in G respectively. We assume that $s_1; s_2; t_1; t_2$ are all distinct. This is without loss of generality, since given an arbitrary instance of 2-DSP, we can introduce new nodes $s_1^0; s_2^0; t_1^0; t_2^0$ such that each s_i^0 has the same neighbors as s_i and each t_i^0 has the same neighbors as t_i (if the input graphs are directed, then s_i^0 and t_i^0 have both the same in-neighbors and the same out-neighbors as s_i and t_i respectively), and then delete the original copies of s_i and t_i. With this addition, the graph contains vertex-disjoint s_i^0 t_i^0 shortest paths if and only if it contains internally vertex-disjoint s_i t_i shortest paths. Moreover, if the graph was originally undirected it stays undirected, and if the graph was originally a DAG it stays a DAG.

We view undirected graphs as graphs whose edges are still ordered pairs of vertices $(u; v)$, but with the property that $(u; v)$ 2 E is an edge if and only if $(v; u)$ 2 E is an edge.

If G is weighted, we let $`(u; v)$ denote the weight of an edge $(u; v)$ $2$ $E$. We assume that edge weights $`(u; v) > 0$ are positive for all $(u; v)$. Given vertices u and v in G, we let $\mathsf{dist}(u; v)$ denote the shortest path distance from u to v in G, i.e., the minimum possible sum of edge weights among all u v paths in G.

Given a path P which passes through vertices u and v in that order, we let $P[u; v]$ denote the u v *subpath* of P. If P is a path in an undirected graph, we let $\overline{P}$ denote the *reverse path* of P, which traverses the vertices of P in reverse order. Given two paths P and Q such that the final vertex of P is the same as the first vertex of Q, we let P Q denote the concatenation of P and Q.

Field Size

We recall the preliminaries from the Finite Field Computation subsection of Section 6.1. In particular we work over a field $\mathsf{F} = \mathsf{F}_{2^q}$. In this chapter, we set q to be the smallest positive integer with

$$2^q \quad 2n^2: \tag{111}$$

Note that $q =$ $(\log n)$.

Topological Order

Any directed acyclic graph (DAG) admits a topological order: this is an ordering $(\ )$ of the vertices in the graph with the property that u v implies that $(v; u)$ is not an edge in the graph. In other words, edges only go forwards with respect to the topological order of a graph. Using depth-first search, it is easy to construct a topological order for a DAG in linear time. This fact will be useful for us.

Proposition 8.4. We can compute a topological order of a DAG in linear time.

See [CLRS09, Section 22.4] for a proof of Proposition 8.4.

Shortest Paths Graphs

For any vertex s in G, we define the *s-shortest paths DAG* to be the directed acyclic graph with the same vertex set as G, which includes an edge $(u; v)$ if and only if $(u; v) \in E$ is the last edge in some $s \rightsquigarrow v$ shortest path in G.

From this definition, it is easy to see that a sequence of vertices P is an $s \rightsquigarrow v$ shortest path in G if and only if P is an $s \rightsquigarrow v$ path in the s-shortest paths DAG of G. Indeed, given a sequence of vertices $P = \langle v_0, \ldots, v_r \rangle$ with $v_0 = s$ and $v_r = v$, if P is an $s \rightsquigarrow v$ shortest path in G, then $\langle v_0, \ldots, v_i \rangle$ is an $s \rightsquigarrow v_i$ shortest path for all $i \in [r]$. This means that (v_{i-1}, v_i) is an edge in the s-shortest paths DAG of G for all $i \in [r]$, so P is an $s \rightsquigarrow v$ path in this DAG as claimed. Conversely, if $P = \langle v_0, \ldots, v_r \rangle$ is an $s \rightsquigarrow v$ path in the s-shortest paths DAG of G, an easy induction argument shows that $\langle v_0, \ldots, v_i \rangle$ is a shortest path in G for all $i \in [r]$. Applying this result for $i = r$ shows that P is an $s \rightsquigarrow t$ shortest path in G as claimed.

The argument in the previous paragraph implies that the s-shortest paths DAG is indeed acyclic, since a cycle cannot be a shortest path in a graph with positive edge weights.

Working with shortest paths DAGs is useful for us because it helps reduce the task of enumerating shortest paths in G to enumerating arbitrary paths in a DAG.

Given a node s in G, we can construct the s-shortest paths DAG of G in linear time, using standard algorithms for computing single-source shortest path distances.

Proposition 8.5 (Single-Source Distances in DAGs)**.** Given a weighted DAG G with distinguished node s, we can compute $\mathsf{dist}(s; v)$ for all vertices v in G in linear time.

See [CLRS09, Section 24.2] for a proof of Proposition 8.5.

Proposition 8.6 (Single-Source Distances in Undirected Graphs)**.** Given a weighted undirected graph G and node s, we can compute $\mathsf{dist}(s; v)$ for all vertices v in G in linear time.

See [Tho97] for a proof of Proposition 8.6.

Proposition 8.7 (Shortest Path DAGs)**.** Let G be a weighted DAG or undirected graph with vertex s. Then we can construct the s-shortest paths DAG of G in linear time.

Proof. By Propositions 8.5 and 8.6, we can compute $\mathsf{dist}(s; v)$ for all vertices v in G in linear time. We claim that an edge $(u; v)$ is in the s-shortest paths DAG of G if and only if

$$\mathsf{dist}(s; v) = \mathsf{dist}(s; u) + \ell(u; v). \tag{112}$$

Indeed, if $(u; v)$ is in the s-shortest paths DAG, it is the last edge on some $s \rightsquigarrow v$ shortest path P in G. Since P is a shortest path, it has length $\mathsf{dist}(s; v)$. Since it has last edge $(u; v)$, its $s \rightsquigarrow u$ prefix must also be a shortest path, so its length also equals $\mathsf{dist}(s; u) + \ell(u; v)$, so eq. (112) holds. Conversely, if eq. (112) holds, then concatenating an $s \rightsquigarrow u$ shortest path in G with the edge $(u; v)$ produces a path of minimum length $\mathsf{dist}(s; v)$. Hence $(u; v)$ is the last edge of an $s \rightsquigarrow v$ shortest path, so $(u; v)$ is in the s-shortest paths DAG as claimed.

So we can construct the s-shortest paths DAG G by going through every edge $(u;v)$ in G, and checking if it satisfies eq. (112). Since we already computed the distances from s to every vertex in G, checking eq. (112) takes $O(1)$ time for each edge. Thus we can construct the s-shortest paths DAG in linear time as claimed.

Additional Notation

For each $i \in [2]$, we define G_i to be the s_i-shortest paths DAG of G.

For each $i \in [2]$ and vertex $v \in V$, we define $V^i_{\text{in}}(v)$ to be the set of in-neighbors of v in G_i, and $V^i_{\text{out}}(v)$ to be the set of out-neighbors of v in G_i. We further let

$$V_{\text{in}}(v) = V^1_{\text{in}}(v) \setminus V^2_{\text{in}}(v) \quad \text{and} \quad V_{\text{out}}(v) = V^1_{\text{out}}(v) \setminus V^2_{\text{out}}(v)$$

be the sets of in-neighbors and out-neighbors respectively of v common to both G_1 and G_2. We also define

$$V_{\text{mix}}(v) = V^1_{\text{in}}(v) \setminus V^2_{\text{out}}(v) \tag{113}$$

to be the *mixed neighborhood* of v, defined to be the set of nodes u such that $(u;v)$ is an edge in G_1 and $(v;u)$ is an edge in G_2.

A pair of paths $\langle P_1; P_2 \rangle$ is a *standard pair* if P_i is an $s_i \rightsquigarrow t_i$ path in G_i for each $i \in [2]$. Equivalently, a standard pair consists of $s_i \rightsquigarrow t_i$ shortest paths P_i in G.

8.3 General Ideas

Throughout this section, G is allowed to be a weighted DAG or undirected graph. All of the arguments in this section apply to both cases.

We will solve 2-DSP using the algebraic framework introduced in Chapter 6.

For every edge $(u;v) \in E$, we introduce an indeterminate variable x_{uv}. We use these variables to enumerate families of pairs of paths in G, following the discussion from the Node-Based subsection of Section 6.2. In particular, each path W is assigned a monomial $\mathbf{x}(W)$ recording the consecutive pairs of nodes it traverses as in eq. (56), and each pair of paths $P = \langle P_1; P_2 \rangle$ is assigned a monomial $\mathbf{x}(P) = \mathbf{x}(P_1; P_2)$ according to eq. (58).

Recall the definition of an *enumerating polynomial* from eq. (59).

Definition 8.8 (Disjoint Paths Polynomial)**.** Let F_{disj} be the enumerating polynomial for the collection of vertex-disjoint standard pairs of paths.

We will solve 2-DSP by testing if the polynomial F_{disj} is nonzero. The following result shows that this is possible, even though we work over a field of characteristic two.

Proposition 8.9 (Testing for Disjoint Paths)**.** The polynomial F_{disj} is nonzero if and only if G contains disjoint $s_i \rightsquigarrow t_i$ shortest paths for $i \in [2]$.

Proof. If G does not have disjoint $s_i \rightsquigarrow t_i$ shortest paths, then F_{disj} is the zero polynomial by definition. If instead G does contain disjoint $s_i \rightsquigarrow t_i$ shortest paths P_i, then F_{disj} contains the monomial $\mathbf{x}(P_1; P_2)$. By looking at the set of variables x_e which appear in this monomial, we can recover the set E of all edges used by the paths P_1 and P_2. Because the P_i are disjoint

s_i t_i paths, P_i is the unique s_i t_i contained in the subgraph of G on E for each $i \in [2]$. Consequently, the monomial $(P_1; P_2)$ uniquely encodes $\hbar P_1; P_2 i$, and no other monomial appearing in F_{disj} cancels it out. So F_{disj} is a nonzero polynomial.

This proves the desired result.

To compute F_{disj}, we need to find a way of enumerating disjoint standard pairs of paths. Doing this directly seems difficult, because it is unclear how to enforce the condition that two paths never intersect. Instead, it might be easier to compute F_{disj} indirectly, by enumerating all standard pairs of paths, and then substracting out those pairs which intersect.

> **Idea 14** Instead of enumerating pairs of disjoint paths directly, we can **count the complement** and enumerate pairs of intersecting paths instead.

Intersecting paths should intuitively be easier to work with, because we can split them along their intersection points into smaller paths.

Before we can enumerate pairs of paths, we should figure how to enumerate collections of individual paths. In Sections 7.2 and 7.3 we saw ways of enumerating paths using matrix inverses and determinants. These methods would take $(n!)$ time however, which is too slow to prove Theorems 8.1 and 8.2. Since we are aiming for linear-time algorithms, we cannot afford to enumerate paths between all pairs of vertices in G. We can however afford to enumerate paths leaving and entering terminals in G.

For each $i \in [2]$ and vertex v, let $L_i(v)$ be the enumerating polynomial for the collection of s_i v paths in G_i. Similarly, let $R_i(v)$ be the enumerating polynomial for the collection of v t_i paths in G_i. The same simple procedure for computing single-source shortest distances in a DAG enables us to efficiently compute the $L_i(v)$ and $R_i(v)$ polynomials.

Lemma 8.10 (Source Paths). For each $i \in [2]$ and vertex v, we have

$$L_i(v) = \sum_{u \in V_{\text{in}}^i(v)} L_i(u) x_{uv}.$$

Proof. Since $L_i(u)$ enumerates s_i u paths in G_i, the polynomial

$$L_i(u) x_{uv}$$

enumerates s_i v paths in G_i whose penultimate vertex is u. Every s_i v path in G_i has some unique penultimate vertex $u \in V_{\text{in}}^i(v)$. Consequently

$$\sum_{u \in V_{\text{in}}^i(v)} L_i(u) x_{uv}$$

enumerates all s_i v paths in G_i, which proves the claim.

Lemma 8.11 (Target Paths). For each $i \in [2]$ and vertex v, we have

$$R_i(v) = \sum_{w \in V_{\text{out}}^i(v)} x_{vw} R_i(v).$$

Proof. Follows from symmetric reasoning to the proof of Lemma 8.10.

Corollary 8.12 (Evaluating Paths). Given an assignment to the x_{uv} variables over $\mathbb{F}$, we can evaluate $L_i(v)$ and $R_i(v)$ at that assignment for all vertices v and $i \in [2]$ in linear time.

Proof. By Proposition 8.4, we can find a topological order $(\quad)$ of G_i in linear time. Let

$$v_1 \qquad\qquad v_r$$

be all the nodes of G occurring after s in the topological order. Let $v_0 = s$.

We can return $L_i(v) = 0$ for all $v \notin v_0, \ldots, v_r$, since any such node v is not reachable from s. We can return $L_i(s) = 1$, since the one-node path $\langle s \rangle$ is assigned weight 1.

For each $j \in [r]$, by Lemma 8.10 we have

$$L_i(v_j) = \sum_{u \in V^i_{\text{in}}(v_j)} L_i(u) x_{uv_j}.$$

We can evaluate $L_i(v_j)$ at the given assignment for $j = 1, \ldots, r$ in order, using the above equation. At iteration j, we compute $L_i(v_j)$ using the values of $L_i(u)$ for $u \quad v_j$ (which will have already been computed since we are proceeding according to the topological order) and the values assigned to the variables x_{uv_j}. It takes $O(\deg_{\text{in}}(v_j))$ time to compute $L_i(v_j)$ in this way for each $j \in [r]$, so the process takes $O(m)$ time overall.

Doing this for each $i \in [2]$ allows us to evaluate all the $L_i(v)$ in linear time. Symmetric reasoning (using Lemma 8.11 and dynamic programming backwards with respect to the topological order) shows that we can also evaluate all the $R_i(v)$ in linear time.

So in linear time, we can compute all the $L_i(v)$ and $R_i(v)$ polynomials, enumerating all shortest paths leaving the sources and entering the targets.

As suggested by Idea 14, we can use these polynomials to reduce the enumeration of disjoint paths to the enumeration of *intersecting* paths.

Definition 8.13 (Intersecting Paths Polynomial). Let $F_\backslash$ be the enumerating polynomial for the collection of intersecting, standard pairs of paths.

Lemma 8.14 (Disjoint Paths Intersecting Paths). We have

$$F_{\text{disj}} = L_1(t_1) L_2(t_2) \quad F_\backslash.$$

Proof. By expanding out the product, we see that $L_1(t_1)L_2(t_2)$ enumerates all standard pairs of paths $\langle P_1, P_2 \rangle$. Each such pair is either vertex-disjoint or consists of paths intersecting at a common node, so we have

$$L_1(t_1)L_2(t_2) = F_{\text{disj}} + F_\backslash$$

which implies the desired result.

The relationship from Lemma 8.14 between F_{disj} and $F_\backslash$ is pictured in Figure 4.

So to compute F_{disj}, it suffices to compute $F_\backslash$.

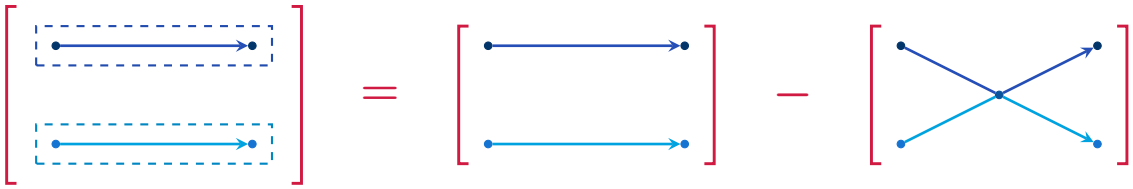

Figure 4: To enumerate the family of disjoint pairs of paths on the left (the dashed borders around the paths indicate that the paths do not intersect), it suffices to enumerate all pairs of paths and subtract out those pairs in the family which intersect at some point.

To compute $F_{\setminus}$, we need to enumerate intersecting, standard pairs of paths. Any standard pair $\langle P_1, P_2 \rangle$ of intersecting paths has a unique *first* intersection point v. Formally, vertex v is defined to be the first node of P_1 which lies in $P_1 \setminus P_2$. The paths uniquely decompose as

$$P_i = A_i \circ B_i,$$

where A_i is an $s_i \rightsquigarrow v$ path and B_i is a $v \rightsquigarrow t_i$ path in G_i for $i \in [2]$, such that A_1 and A_2 intersect only at v.

We say the pair of paths $\langle A_1, A_2 \rangle$ forms a *linkage*, because they link the terminal nodes to a common endpoint v, but are disjoint otherwise.

Definition 8.15 (Source Linkages). Given a vertex v, let $S(v)$ be the collection of pairs of paths $\langle P_1, P_2 \rangle$ such that each P_i is an $s_i \rightsquigarrow v$ path in G_i, and $P_1 \setminus P_2 = \{v\}$. Let $S(v)$ be the enumerating polynomial for $S(v)$.

Since linkages naturally arise when we decompose intersecting pairs of paths, it seems like enumerating linkages should help compute $F_{\setminus}$. As with F_{disj} however, it is not immediately clear how to enumerate linkages, because the condition that they are internally vertex-disjoint is a general "global" condition that seems difficult to enforce.

We can try repeating the strategy of Idea 14, and compute $S(v)$ by first enumerating all pairs of paths $\langle P_1, P_2 \rangle$ where P_i is an $s_i \rightsquigarrow v$ path in G_i, and then subtracting out those where the P_i intersect at some node before v. The P_i paths could intersect in fairly complicated ways however—how can we enumerate all such intersecting pairs?

The key idea we can use at this point is that because we are working over a field $\mathbb{F}$ of characteristic two, many pairs of intersecting paths actually cancel out in our enumeration. Indeed, as above consider a pair of paths $\langle P_1, P_2 \rangle$ such that P_i is an $s_i \rightsquigarrow v$ path in G_i, such that P_1 and P_2 intersect at some node u before v. If the subpaths $P_1[u, v]$ and $P_2[u, v]$ are distinct, then we can form a new pair of paths $\langle Q_1, Q_2 \rangle$ by

$$Q_1 = P_1[s_1, u] \circ P_2[u, v] \quad \text{and} \quad Q_2 = P_2[s_2, u] \circ P_1[u, v]$$

swapping the $u \rightsquigarrow v$ subpaths of P_1 and P_2. Since the edges traversed by the Q_1 are identical to the edges traversed by the P_i, both pairs of paths will have the same weight $w(P_1, P_2) = w(Q_1, Q_2)$. Then modulo two, the weights $w(P_1, P_2)$ and $w(Q_1, Q_2)$ will cancel out when we enumerate these pairs of paths. This means that when we are subtracting out pairs of paths that intersect somewhere before v (in order to compute $S(v)$), we do not need to explicitly handle those pairs that intersect at a node u and have different $u \rightsquigarrow v$ subpaths,

because the contributions from such pairs will cancel over $\mathbb{F}$ anyway. We only need to worry about subtracting out those paths which intersect along a full segment leading up to v, which should intuitively be easier to handle, since such pairs of paths are quite structured.

> **Idea 15** When we enumerate modulo two, the only intersecting pairs of paths which survive are those that overlap on a single segment. In particular, the contributions from the more complicated intersecting pairs of paths vanish for free.

We note that the general phenomenon of disjoint path structures simplifying modulo two has been used in algorithms many times before [BHT12, BH19, BHK22]. This simplification appears to be connected to the idea of permanents reducing to determinants modulo two, discussed at the beginning of Chapter 6.

Based off the intuition above, we next introduce some lemmas which help identify common situations where enumerating polynomials simplify modulo two. The main idea is that enumerating polynomials for a family $\mathcal{F}$ can be simplified whenever we can identify a subfamily $\mathcal{V} \subseteq \mathcal{F}$ of pairs of paths which admits a nice *subpath swapping* involution.

Subpath Swapping

Lemma 8.16 (Shortest Path Swapping). Let P_1 and P_2 be shortest paths passing through vertices a and b in that order. Then the walks obtained by swapping the $a \to b$ subpaths in P_1 and P_2 are also shortest paths.

Proof. Since P_1 and P_2 are shortest paths, each of their $a \to b$ subpaths have length $\mathsf{dist}(a, b)$. Since these subpaths have the same length, swapping these subpaths of P_1 and P_2 produce walks with the same endpoints and lengths as P_1 and P_2 respectively. Since P_1 and P_2 are shortest paths, this implies that the new walks are shortest paths as well. Here, we are using the fact that we work over graphs with positive edge weights, so that any walk whose length equals to the shortest path distance between its endpoints cannot have repeat vertices.

Lemma 8.16 says that swapping subpaths between shortest paths is a "safe operation," in the sense after the subpaths swap we are still working with shortest paths. We will use this observation to argue correctness for the subpath swapping argument we informally discussed in the paragraph before Idea 15.

The following lemma describes a general setting in which we can simplify enumerating polynomials for pairs of paths modulo two, by matching up different pairs via subpaths swaps. We will make use of it frequently when proving correctness of our 2-DSP algorithms.

Lemma 8.17 (Vanishing Modulo 2). Let $\mathcal{F}$ be a family of pairs of paths in G, and let $\mathcal{V} \subseteq \mathcal{F}$. Suppose there exist maps $\alpha, \beta : \mathcal{V} \to V$ and $\tau : \mathcal{V} \to \mathcal{V}$ such that for all $P = \langle P_1, P_2 \rangle \in \mathcal{V}$,

1. the vertices $a = \alpha(P)$ and $b = \beta(P)$ lie in $P_1 \setminus P_2$, a appears before b in P_1 and P_2, and the subpaths $P_1[a, b]$ and $P_2[a, b]$ are distinct;
2. $\tau(P) = \langle Q_1, Q_2 \rangle$, where Q_1 is obtained by replacing the $a \to b$ subpath in P_1 with $P_2[a, b]$, and Q_2 is obtained by replacing the $a \to b$ subpath in P_2 with $P_1[a, b]$; and
3. we have $\tau(\tau(P)) = P$.

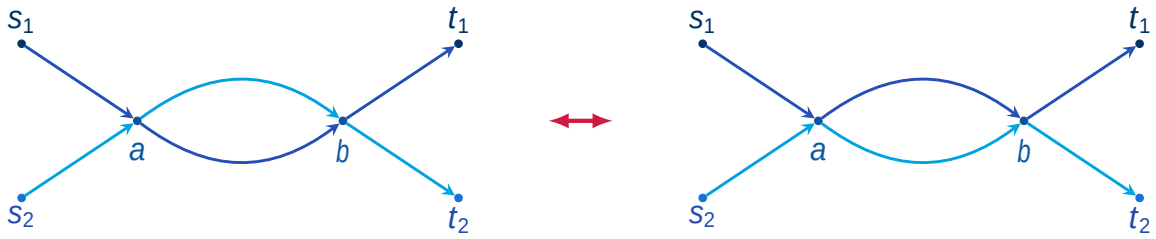

Figure 5: Given paths P_1 and P_2 which intersect at nodes $a = (P_1; P_2)$ and $b = (P_1; P_2)$, such that a appears before b on both paths, if we swap the a to b subpaths of of P_1 and P_2 to produce new paths Q_1 and Q_2 respectively, then these pairs $(P_1; P_2) = (Q_1; Q_2)$ have the same monomials. Moreover, swapping the a to b subpaths of Q_1 and Q_2 recovers P_1 and P_2.

Then the enumerating polynomial for F is the same as the enumerating polynomial for $F \cap V$.

Proof. Let F be the enumerating polynomial for F. By definition, we have

$$F = \sum_{P \in F} (P) = \sum_{P \in F \cap V} (P) + \sum_{P \in V} (P). \tag{114}$$

Take any $P = \langle P_1; P_2 \rangle \in V$. By property 1 from the lemma statement, the subpaths from (P) to (P) in P_1 and P_2 are distinct. Then by property 2, $(P) \neq P$. Consequently, by property 3, we can partition $V = V_1 \sqcup V_2$ into two equally sized pieces such that is a bijection from V_1 to V_2. So we can write

$$\sum_{P \in V} (P) = \sum_{P \in V_1} (P) + \sum_{P \in V_2} (P) = \sum_{P \in V_1} ((P) + ((P))). \tag{115}$$

By property 2, the multiset of edges traversed by the pair P is the same as the multiset of edges traversed by (P), for all $P \in V$. Consequently, $(P) = ((P))$ for all $P \in V$.

The subpath swapping procedure determined by is depicted in Figure 5.

Since we work over a field of characteristic two, this implies that

$$\sum_{P \in V_1} ((P) + ((P))) = 0.$$

Substituting the above equation into eq. (115) implies that

$$\sum_{P \in V} (P) = 0.$$

Then substituting the above equation into eq. (114) yields

$$F = \sum_{P \in F \cap V} (P).$$

This proves that F is the enumerating polynomial for $F \cap V$ as desired.

In general, Lemma 8.17 is useful in situations where we have a "complicated" family of pairs of paths F we need to enumerate. If we can identify a subfamily $V \subseteq F$ for which there exists maps $\ ;\ ;\ $ satisfying the conditions from the statement of Lemma 8.17 (intuitively, these maps describe a way of matching up pairs of paths in V with equal weight), then Lemma 8.17 says to enumerate F it suffices to enumerate the "simpler" family $F \cap V$.

Having established these subpath swapping lemmas, we return to the task of enumerating the collection of source linkages $S(v)$ from Definition 8.15. Using Idea 15, we argue that instead of enumerating $S(v)$, it suffices to enumerate a simpler collection where the disjointness condition of linkages is *relaxed*.

Definition 8.18 (Relaxed Source Linkage)**.** Given a vertex v, let $\widetilde{S}(v)$ be the collection of pairs of paths $\langle P_1, P_2 \rangle$, where each P_i is an $s_i \rightsquigarrow v$-path in G_i, and the penultimate vertices of P_1 and P_2 are distinct.

So while a pair of paths in $S(v)$ cannot overlap at *any vertex* before v, a pair of paths in the collection $\widetilde{S}(v)$ may overlap before v, as long as the paths in this pair do not overlap *immediately* before v. The next lemma shows that modulo two, the enumerating polynomials for $S(v)$ and $\widetilde{S}(v)$ are the same. Recall from Definition 8.15 that we let $S(v)$ denote the enumerating polynomial for $S(v)$.

Lemma 8.19. For any vertex v, the polynomial $S(v)$ enumerates $\widetilde{S}(v)$.

Proof. For convenience, let $F = \widetilde{S}(v)$. Let

$$V = F \cap S(v)$$

be the family of pairs of paths $\langle P_1, P_2 \rangle$ where each P_i is an $s_i \rightsquigarrow v$ path in G_i, such that

1. the paths P_1 and P_2 intersect at some vertex other than v, and
2. the vertices immediately before v on P_1 and P_2 are distinct.

Take arbitrary $\langle P_1, P_2 \rangle \in V$. Let u be the vertex in $P_1 \setminus P_2$ maximizing the value of $\mathsf{dist}(u, v)$. By condition 1 above, $u \neq v$. By condition 2 above, $P_1[u, v]$ and $P_2[u, v]$ are distinct.

Now define walks

$$Q_1 = P_1[s_1, u] \circ P_2[u, v] \quad \text{and} \quad Q_2 = P_2[s_2, u] \circ P_1[u, v].$$

By Lemma 8.16 each Q_i is a shortest path, and thus an $s_i \rightsquigarrow v$ path in G_i.

The pair $\langle Q_1, Q_2 \rangle$ satisfies condition 1 above, since $u \in Q_1 \setminus Q_2$. This pair also satisfies condition 2 above, since the penultimate vertices of Q_1 and Q_2 are the same as the penultimate vertices of P_2 and P_1 respectively. Also, u is the node in $Q_1 \setminus Q_2$ maximizing $\mathsf{dist}(u, v)$, so applying the same subpath swapping operation from above to $\langle Q_1, Q_2 \rangle$ produces $\langle P_1, P_2 \rangle$.

By the discussion in the previous paragraph, the map $\ $ sending $\langle P_1, P_2 \rangle$ to node u, the map $\ $ sending $\langle P_1, P_2 \rangle$ to node v, and the map $\ $ sending $\langle P_1, P_2 \rangle$ to $\langle Q_1, Q_2 \rangle$ meet the conditions of Lemma 8.17, so the enumerating polynomial for F is the same as the enumerating polynomial for $F \cap V = S(v)$. The enumerating polynomial for $S(v)$ is $S(v)$, so this proves the desired result.

By Lemma 8.19, instead of trying to enumerate $S(v)$ directly, which seems difficult, we can instead focus on enumerating $\widetilde{S}(v)$, whose conditions seems easier to handle. The high-level strategy being employed is summarized in the idea below.

> **Idea 16** When enumerating pairs of paths modulo two, we can **replace global conditions** about paths not intersecting anywhere outside a node v **with simpler local conditions** about paths not intersecting immediately before or after v.

Lemma 8.20 (Enumerating Source Linkages)**.** For each vertex v, we have

$$S(v) = L_1(v)L_2(v) \sum_{u \in V_{\text{in}}(v)} L_1(u)L_2(u)x_{uv}^2.$$

Proof. To prove the lemma, we first establish the following claim.

Claim 8.21. For any vertices u and v with $u \in V_{\text{in}}(v)$, the polynomial

$$L_1(u)L_2(u)x_{uv}^2 \tag{116}$$

enumerates the collection of pairs of paths $\langle P_1, P_2 \rangle$ such that each P_i is an $s_i \to v$ path with final edge (u, v).

Proof. Let $\langle P_1, P_2 \rangle$ be a pair of paths where each P_i is an $s_i \to v$ path with final edge (u, v). Then we can split the P_i along their final edges as

$$P_1 = P_1[s_1, u] \to (u, v) \quad \text{and} \quad P_2 = P_2[s_2, u] \to (u, v).$$

The paths $P_i[s_i, u]$ are enumerated by the $L_i(u)$ factors in eq. (116), and the two copies of (u, v) are encoded by the x_{uv}^2 factor in eq. (116). Conversely, any monomial in the expansion of eq. (116) is the product of monomials for some $s_i \to u$ paths Q_i in G_i and two occurrences of the edge (u, v), so that if we define

$$P_1 = Q_1 \to (u, v) \quad \text{and} \quad P_2 = Q_2 \to (u, v)$$

then the monomial we are considering is precisely (P_1, P_2). This proves the claim. Note that we used the fact that $u \in V_{\text{in}}(v)$ to ensure that (u, v) is an edge in both G_1 and G_2. □

By expanding out the definitions of $L_1(v)$ and $L_2(v)$ in the product product, we see that

$$L_1(v)L_2(v)$$

enumerates all pairs of paths $\langle P_1, P_2 \rangle$ where P_i is an $s_i \to v$ path in G_i. By summing the enumerating polynomials from Claim 8.21 for each $u \in V_{\text{in}}(v)$, we see that

$$\sum_{u \in V_{\text{in}}(v)} L_1(u)L_2(u)x_{uv}^2$$

enumerates all pairs of paths $\langle P_1, P_2 \rangle$ where P_i is an $s_i \to v$ path in G_i, and the penultimate vertices of the P_i are the same.

Figure 6: To enumerate the family of intersecting pairs of paths on the left, we can perform casework on the earliest intersection point v for the paths (the dashed border on the subpaths on the right indicates that the paths do not intersect before v).

By subtracting the polynomials from the two previous equations, we get that

$$L_1(v)L_2(v) \quad \sum_{u2V_{\text{in}}(v)} L_1(u)L_2(u)x_{uv}^2$$

enumerates all pairs of paths $\langle P_1; P_2 \rangle$ where each P_i is an s_i v path in G_i, and the penultimate vertices of the P_i are distinct.

Consequently, the above equation is the enumerating polynomial for $\widetilde{S}(v)$. By Lemma 8.19, this implies that

$$S(v) = L_1(v)L_2(v) \quad \sum_{u2V_{\text{in}}(v)} L_1(u)L_2(u)x_{uv}^2.$$

as desired.

In the next section, we use our construction from Lemma 8.20 of the enumerating polynomial for $S(v)$ to construct F_{disj} when G is a weighted DAG.

8.4 Directed Acyclic Graphs

In this section, we assume that G is a weighted DAG, and fix a topological order $(\ )$ of G.

As noted in Lemma 8.14, to compute F_{disj} it suffices to compute $F_{\setminus}$, the enumerating polynomial for intersecting, standard pairs of paths. We perform this enumeration by casework on the first intersection v of the pair of paths. Here "first intersection" means that v is the earliest node in both paths with respect to the topological order of G. We then (implicitly) use Lemma 8.19 to relax the global condition that the pair of paths have *first* intersection at v to the simpler local condition that the pair intersects at v, and does not intersect at the node immediately before v.

This overall strategy is depicted in Figures 6 and 7.

Lemma 8.22 (Enumerating Intersecting Paths). We have

$$F_{\setminus} = \sum_{v2V} S(v)R_1(v)R_2(v).$$

Proof. The lemma follows from the following claim.

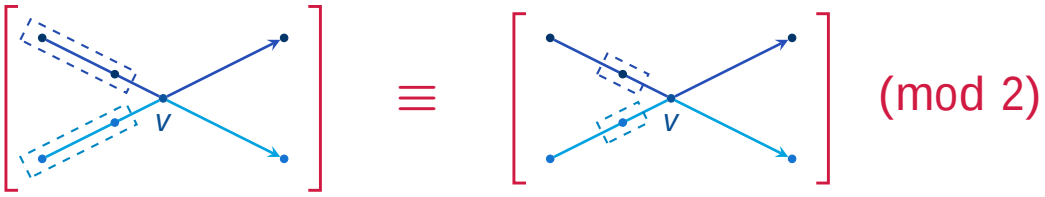

Figure 7: If we work modulo two, then we can enumerate pairs of paths which have common first intersection at node v by enumerating pairs of paths which intersect at v and have the property that the vertices appearing immediately before v on each path are distinct.

B Claim 8.23. For any vertex v, the polynomial

$$S(v)R_1(v)R_2(v) \tag{117}$$

enumerates the standard pairs $\langle P_1; P_2 \rangle$ such that P_1 and P_2 have first intersection at v.

Proof. Let $\langle P_1; P_2 \rangle$ be a standard pair with first intersection at v. Then we can decompose

$$P_i = P_i[s_i; v] \quad P_i[v; t_i]$$

for each $i \in [2]$, and observe that the pair $\langle P_1[s_1; v]; P_2[s_2; v] \rangle$ is enumerated by the $S(v)$ factor in eq. (117), while the $P_i[v; t_i]$ paths are enumerated by the respective $R_i(v)$ factors from eq. (117).

Conversely, any monomial in the expansion of eq. (117) is the product of a monomial from $S(v)$ with monomials from $R_i(v)$ for $i \in [2]$. Any monomial from $S(v)$ is of the form $(A_1; A_2)$, where each A_i is an $s_i \quad v$ path in G_i such that A_1 and A_2 only intersect at v. For each $i \in [2]$, any monomial from R_i is of the form (B_i), where B_i is a $v \quad t_i$ path in G_i. Then if we define

$$P_i = A_i \quad B_i$$

we see that the P_i are $s_i \quad t_i$ paths in G_i with the property that P_1 and P_2 have first intersection at v. Here, we are using the fact that G is a DAG—this ensures that every node in A_1 or A_2 appears at or before v in the topological order, and that every node in B_1 or B_2 appears at or after v in the topological order, so that $A_1 \setminus B_2 = A_2 \setminus B_1 = \{v\}$. □

Since G is a DAG, every intersecting pair of paths intersects at some unique earlier vertex. Then by Claim 8.23, the polynomial

$$\sum_{v \in V} S(v)R_1(v)R_2(v)$$

enumerates all intersecting, standard pairs of paths $\langle P_1; P_2 \rangle$. This proves the lemma.

Having established a formula for $F_{\setminus}$ in Lemma 8.22, we present our algorithm for solving 2-DSP in weighted DAGs in Algorithm 9. Note that in the algorithm, we never compute polynomials explicitly, and instead compute polynomial evaluations with respect to the random assignment in step 2 of Algorithm 9.

Lemma 8.24. Algorithm 9 solves 2-DSP in weighted DAGs with high probability.

Algorithm 9. The 2-DSP Algorithm in Directed Acyclic Graphs

Inputs: A directed acyclic graph G, with specified sources $s_1; s_2$ and targets $t_1; t_2$.
Returns: YES if G contains disjoint s_i t_i shortest paths for i 2 [2], NO otherwise.

1. For each i 2 [2], compute the s_i-shortest paths DAG G_i.
2. Assign independent, uniform random values x_{uv} from F for each $(u; v)$ 2 E.
3. For every vertex v and each i 2 [2], compute $L_i(v)$ and $R_i(v)$.
4. For every vertex v, compute
$$S(v) = L_1(v)L_2(v) \quad \sum_{u2V_{\text{in}}(v)} L_1(u)L_2(u)x_{uv}^2:$$
5. Compute
$$F_{\setminus} = \sum_{v2V} S(v)R_1(v)R_2(v):$$
6. Compute
$$F_{\text{disj}} = L_1(t_1)L_2(t_2) \quad F_{\setminus}:$$
Return YES if F_{disj} is nonzero, NO if F_{disj} is zero.

Proof. By Lemma 8.20, step 4 of Algorithm 9 correctly computes $S(v)$ for each vertex v.

By Lemma 8.22, step 5 of Algorithm 9 correctly computes $F_{\setminus}$.

By Lemma 8.14, step 6 of Algorithm 9 correctly computes F_{disj}.

By Proposition 8.9, F_{disj} is a nonzero polynomial if and only if G contains vertex-disjoint s_i t_i shortest paths for i 2 [2]. By definition, F_{disj} is a polynomial of degree at most $2n$. Then by Proposition 6.1 and our choice of q in eq. (111), with high probability the evaluation of F_{disj} on the random assignment from step 2 of Algorithm 9 is nonzero if and only if G contains a solution to the 2-DSP problem. Thus with high probability, Algorithm 9 returns the correct answer to the 2-DSP problem in step 6.

Proof of Theorem 8.2. By Lemma 8.24, Algorithm 9 solves the 2-DSP problem in weighted DAGs. It remains to prove that we can implement Algorithm 9 to run in linear time.

Step 1 of Algorithm 9 takes linear time by Proposition 8.7.

Step 2 of Algorithm 9 takes linear time because we spend $O(1)$ time at each edge of G.

Step 3 of Algorithm 9 takes linear time by Corollary 8.12.

For each fixed vertex v, computing $S(v)$ using the formula in step 4 of Algorithm 9 takes $O(\deg_{\text{in}}(v))$ time. Summing this runtime bound over all vertices v, we see that step 4 of Algorithm 9 takes $O(m)$ time.

Step 5 of Algorithm 9 takes $O(n)$ time since we add and multiply $O(n)$ terms.

Step 6 of Algorithm 9 takes $O(1)$ time given our previous computations.

So overall Algorithm 9 runs in linear time as claimed.

8.5 Undirected Graphs

In this section, we assume that G is a weighted undirected graph.

Our goal is to solve 2-DSP in weighted undirected graphs. We continue the approach begun in Section 8.3, and aim to compute $F_{\setminus}$ efficiently, by enumerating intersecting, standard pairs of paths. As pictured in Figure 6, we will perform this enumeration by casework on the first intersection v of the pairs of paths $\langle P_1; P_2 \rangle$. By "first intersection" we mean that v is the first vertex of P_1 lying in $P_1 \setminus P_2$.

In the case of DAGs, we were able to use the simple formula from Claim 8.23 to enumerate standard pairs of paths with first intersection v, because the topological order ensured that v was also the first vertex of P_2 lying in $P_1 \setminus P_2$. This no longer holds in undirected graphs, as shown in Figure 8 for example.

Even though this property no longer holds, the P_i are shortest paths, and so intuitively cannot intersect in an arbitrary manner—there should still be useful structure we can impose on the ways P_1 and P_2 can overlap.

Shortest Paths Structure

The following observation will help us constrain how shortest paths can overlap in G.

Proposition 8.25 (Shortest Path Orderings)**.** Let G be a weighted undirected graph. Suppose vertices $a; b; c$ appear in that order on some shortest path of G. Then on any shortest path in G passing through these three vertices, b appears between a and c.

Proof. Since some shortest path in G passes through vertices $a; b; c$ in that order, we know that $\mathsf{dist}(a; b)$ and $\mathsf{dist}(b; c)$ are both less than $\mathsf{dist}(a; c)$:

Now, consider any shortest path in G which passes through these three vertices.

If a appears between b and c in this shortest path, then

$$\mathsf{dist}(a; c) < \mathsf{dist}(b; c)$$

which contradicts the observation from the first sentence.

Similarly, if c appears between a and b in this shortest path, then

$$\mathsf{dist}(a; c) < \mathsf{dist}(a; b)$$

which again contradicts the observation from the first sentence of this proof.

Note that in the arguments above, we used the facts that $\mathsf{dist}(u; v) = \mathsf{dist}(v; u)$ for any vertices u and v in an undirected graph, and that G has positive edge weights.

Thus b must appear between a and c on the shortest path as claimed.

Using Proposition 8.25, we can classify the ways two shortest paths can overlap.

Definition 8.26 (Intersection Types)**.** Let P_1 and P_2 be intersecting shortest paths in a weighted undirected graph. Let v be the first vertex in P_1 lying in $P_1 \setminus P_2$.

Figure 8: The unweighted undirected graph G depicted on the left has unique s_i t_i shortest paths P_i, for i 2 [2]. As shown on the right, the first vertex on P_1 lying in $P_1 \setminus P_2$ is node v, which is distinct from w, the first vertex on P_2 lying in $P_1 \setminus P_2$. Here P_1 and P_2 are *reversing* paths, as defined in Definition 8.26. Reversing paths can appear in undirected graphs, but not in DAGs.

- If $P_1 \setminus P_2 = fvg$, we say P_1 and P_2 have *single intersection*.
- If $jP_1 \setminus P_2j$ 2 and v is also the first vertex of P_2 in $P_1 \setminus P_2$, we say P_1 and P_2 *agree*.
- If $jP_1 \setminus P_2j$ 2 and v is the last vertex of P_2 in $P_1 \setminus P_2$, we say P_1 and P_2 are *reversing*.

If paths P_1 and P_2 do not agree, we say they *disagree*, i.e., have single intersection or are reversing. Equivalently, P_1 and P_2 disagree if the first vertex v in P_1 lying in $P_1 \setminus P_2$ is also the last vertex in P_2 lying in $P_1 \setminus P_2$.

Lemma 8.27 (Intersection Types are Exhaustive). Let P_1 and P_2 be intersecting shortest paths in a weighted undirected graph. Then P_1 and P_2 either agree, are reversing, or have single intersection.

Proof. If $jP_1 \setminus P_2j = 1$, then the paths have single intersection.

Otherwise, $jP_1 \setminus P_2j$ 2. Let v be the first vertex in P_1 lying in $P_1 \setminus P_2$. To prove the lemma, it suffices to show that v is either the first or last vertex in P_2 lying in $P_1 \setminus P_2$.

Suppose to the contrary that v is not the first or last vertex of P_2 in $P_1 \setminus P_2$.

Let u and w be the first and last vertices respectively in path P_2 appearing in $P_1 \setminus P_2$. By assumption, $u; v; w$ are all distinct. By definition, P_2 passes through $u; v; w$ in that order. Since P_2 is a shortest path, by Proposition 8.25, v must appear between u and w on P_1. This contradicts the definition of v as the first vertex on P_1 lying in the intersection $P_1 \setminus P_2$.

Thus our initial assumption was false, and the paths P_1 and P_2 must agree or be reversing whenever $jP_1 \setminus P_2j$ 2, which proves the desired result.

By Lemma 8.27, we can enumerate intersecting, standard pairs of paths $hP_1; P_2i$ by casework on the type of intersection they have, from the options listed in Definition 8.26. One issue we run into if we attempt to use this strategy, is that it is not clear how to enumerate standard pairs with single intersection in linear time.

Indeed, fix a vertex v. Suppose we want to enumerate standard pairs $hP_1; P_2i$ with single intersection at v. Let a_i and b_i denote the vertices appearing on P_i immediately before and after v respectively. Using subpath swapping arguments and the strategy of Idea 16, we can show that this enumeration is equivalent, modulo two, to the task of enumerating standard pairs $hP_1; P_2i$ such that $a_1 \in fb_1; b_2g$ and $b_1 \in fa_1; a_2g$. The natural way to enumerate such pairs for fixed v is to use the principle of inclusion-exclusion on the conditions for the vertices a_1 and b_1, which can take $(\deg_{in}(v) \deg_{out}(v))$ time in general (intuitively because

in one term of the inclusion-exclusion expansion, we would need to enumerate the case where $a_1 = b_1$ and $a_2 = b_2$, and in this situation a_1 could be any vertex in $V_{\text{in}}(v)$ and a_2 could be any vertex in $V_{\text{out}}(v)$). Doing this for all vertices v would then take (mn) time, which is too slow for our purposes.

The takeaway from the previous paragraph is that naively enumerating intersecting, standard pairs of paths for the intersection types from Definition 8.26 separately does not seem to yield a linear-time algorithm for computing $F_{\setminus}$.

However, it turns out that a minor change to this approach does work. Specifically, if we bundle the pairs which are reversing or have single intersection into a single group of the disagreeing, standard pairs of paths, this collection can be enumerated in linear time using the idea of Idea 16. Similarly, the remaining group of agreeing, standard pairs of paths can be enumerated efficiently.

We introduce polynomials enumerating these collections of paths.

Definition 8.28 (Agreeing Paths Polynomial)**.** Let F_{agree} be the enumerating polynomial for the collection of standard pairs of paths which agree.

Definition 8.29 (Disagreeing Paths Polynomial)**.** Let F_{dis} be the enumerating polynomial for the collection of standard pairs of paths which disagree.

Lemma 8.30. We have

$$F_{\setminus} = F_{\text{agree}} + F_{\text{dis}}.$$

Proof. This follows immediately from the fact that any intersecting shortest paths either agree or disagree, by Lemma 8.27.

Motivated by Lemma 8.30, we next focus on computing F_{agree} and F_{dis} efficiently.

Agreeing Paths

In this subsection, we show how to compute F_{agree} efficiently.

Any shortest paths P_1 and P_2 which agree have a common first intersection by definition. The next lemma shows that they these paths also have a common final intersection. Intuitively, this is because agreeing paths have a consistent shortest path ordering.

Lemma 8.31 (Common Final Intersection)**.** If paths P_1 and P_2 agree, then they have a common last intersection point, distinct from their common first intersection point.

Proof. Since P_1 and P_2 agree, they have a common first intersection at some vertex v. Then the vertex $w \in P_1 \setminus P_2$ which maximizes $\text{dist}(v, w)$ must be the last node on both P_1 and P_2 in $P_1 \setminus P_2$. Since the paths agree, we have $|P_1 \setminus P_2| \geq 2$, so $w \neq v$.

From the intuition of Idea 15, the only agreeing, standard pairs of paths with first intersection v, that have nonzero contribution when we are enumerating modulo two should be those pairs which overlap at a segment beginning at vertex v. In particular, we should only have to enumerate those pairs which overlap at some edge (v, w) exiting v.

Next, we introduce the collection of pairs of paths satisfying the properties discussed in the previous paragraph, and then prove that F_{agree} enumerates this collection.

Definition 8.32 (Edge-Agreeing). We say a pair of paths $\langle P_1; P_2 \rangle$ is *edge-agreeing* if each P_i is an s_i t_i path in G_i, and P_1 and P_2 traverse a common edge in the same direction.

Since G_i is the s_i-shortest paths DAG of G, the paths in an edge-agreeing pair are always shortest paths. In fact, an edge-agreeing pair is always a standard pair.

Lemma 8.33 (Edge-Agreeing Agreeing). Any edge-agreeing pair of paths is agreeing.

Proof. Suppose a pair of paths $\langle P_1; P_2 \rangle$ is edge-agreeing. Let $e = (a; b)$ be an edge traversed by both P_1 and P_2 (by definition, such an edge exists).

Since $\{a; b\}$ $P_1 \setminus P_2$, we have $|P_1 \setminus P_2|$ 2.

Then by Lemma 8.27, P_1 and P_2 are either agreeing or reversing.

Suppose to the contrary that P_1 and P_2 are reversing paths. Let v be the first vertex in path P_1 lying in $P_1 \setminus P_2$. Then $v \neq b$, since a appears before b on P_1.

Since the paths are reversing, v is the final vertex in P_2 lying in $P_1 \setminus P_2$. Then $v \neq a$, since b appears after a on P_2. Path P_1 passes through vertices $v; a; b$ in that order. Since the P_i are s_i t_i paths in G_i, the P_i are shortest paths. Then by Proposition 8.25, a must appear between v and b on any shortest path containing these three vertices. However, this contradicts the fact that b is between a and v on P_2.

Thus P_1 and P_2 are not reversing, and so must agree as claimed.

Next, we invoke Lemma 8.17 to show that the enumerating polynomials for agreeing, standard pairs of paths and edge-agreeing pairs are the same.

Lemma 8.34. The polynomial F_{agree} enumerates the family of edge-agreeing pairs of paths.

Proof. Let F be the collection of agreeing, standard pairs of paths $\langle P_1; P_2 \rangle$.

Let F_{edge} be the family of edge-agreeing pairs of paths.

By Lemma 8.33, we have F_{edge} F.

Let $V = F \cap F_{\text{edge}}$ be the collection of pairs of paths which agree but are not edge-agreeing.

Take arbitrary $\langle P_1; P_2 \rangle$ 2 V. Since $\langle P_1; P_2 \rangle$ is agreeing, P_1 and P_2 have a unique first intersection v. By Lemma 8.31, these paths also have a unique last intersection $w \neq v$.

Hence, we can decompose the paths into

$$P_1 = P_1[s_1; v] \quad P_1[v; w] \quad P_1[w; t_1] \quad \text{and} \quad P_2 = P_2[s_2; v] \quad P_2[v; w] \quad P_2[w; t_2]:$$

Now define walks

$$Q_1 = P_1[s_1; v] \quad P_2[v; w] \quad P_1[w; t_1] \quad \text{and} \quad Q_2 = P_2[s_2; v] \quad P_1[v; w] \quad P_2[w; t_2]$$

by swapping the v w subpaths of P_1 and P_2.

By Lemma 8.16 each Q_i is a shortest path, and thus an s_i t_i path in G_i.

Since P_1 and P_2 are not edge-agreeing, the subpaths $P_1[v; w]$ and $P_2[v; w]$ are distinct.

The s_i v subpaths of each Q_i and P_i are the same, so Q_1 and Q_2 have common first intersection at v. Hence paths Q_1 and Q_2 agree, so $\langle Q_1; Q_2 \rangle$ 2 F.

Since $\langle P_1; P_2 \rangle$ is not edge-agreeing, neither is $\langle Q_1; Q_2 \rangle$. So $\langle Q_1; Q_2 \rangle$ 62 F_{edge}.

Since $\langle Q_1; Q_2 \rangle$ is in F but not in F_{edge}, we have $\langle Q_1; Q_2 \rangle$ 2 V.

Since the w t_i subpaths of each Q_i and P_i are the same, Q_1 and Q_2 have common last intersection at w. So the common first intersection v and common last intersection w are the same for the pairs $\langle P_1; P_2 \rangle$ and $\langle Q_1; Q_2 \rangle$. Moreover, swapping the v w subpaths of paths Q_1 and Q_2 recovers paths P_1 and P_2 respectively.

The above discussion implies that the map sending $\langle P_1; P_2 \rangle$ to node v, the map sending $\langle P_1; P_2 \rangle$ to node w, and the map sending $\langle P_1; P_2 \rangle$ to $\langle Q_1; Q_2 \rangle$ meet the conditions of Lemma 8.17, so the enumerating polynomial for F is the same as the enumerating polynomial for $F \cap V = F_{\text{edge}}$. The enumerating polynomial for F is F_{agree}.

Thus the enumerating polynomial for F_{edge} is F_{agree} as well, proving the claim.

We now use the polynomials introduced in Section 8.3 to construct a formula for F_{agree}. The formula is similar to the expression for $F_{\setminus}$ over weighted DAGs presented in Lemma 8.22. Intuitively this makes sense, because agreeing pairs of shortest paths have a common first intersection, and so behave like shortest paths in DAGs.

Lemma 8.35 (Enumerating Agreeing Pairs). We have

$$F_{\text{agree}} = \sum_{v \in V} \sum_{w \in V_{\text{out}}(v)} \left(S(v) x_{vw}^2\right) R_1(w) R_2(w).$$

Proof. Let F be the family of edge-agreeing pairs. By Lemma 8.34, it suffices to show that

$$\sum_{v \in V} \sum_{w \in V_{\text{out}}(v)} \left(S(v) x_{vw}^2\right) R_1(w) R_2(w)$$

is the enumerating polynomial for F. To that end, the following claim about the individual terms of the above sum will be useful.

B Claim 8.36. For any choice of vertices v and w with $w \in V_{\text{out}}(v)$, the polynomial

$$\left(S(v) x_{vw}^2\right) R_1(w) R_2(w) \tag{118}$$

enumerates all standard pairs of paths $\langle P_1; P_2 \rangle$ such that P_1 and P_2 overlap at edge $e = (v; w)$, and have common first intersection at v.

Proof. Take any pair $\langle P_1; P_2 \rangle$ satisfying the conditions from the statement of the claim. Then we can decompose the P_i paths into

$$P_i = P_i[s_i; v] \quad (v; w) \quad P_i[w; t_i]$$

such that the $P_i[s_i; v]$ subpaths intersect only at v.

This means that the pair $\langle P_1[s_1; v]; P_2[s_2; v] \rangle$ is enumerated by $S(v)$, the two edges $(v; w)$ are enumerated by x_{vw}^2, and each path $P_i[w; t_i]$ is enumerated by $R_i(w)$, so that the expansion of the polynomial in eq. (118) includes the monomial $(P_1; P_2)$.

Conversely, any monomial in the expansion of eq. (118) is equal to the product of

- a monomial $(A_1; A_2)$ recording the edges traversed by some pair of paths $\langle A_1; A_2 \rangle$ only intersecting at node v, where A_i is an s_i v path in G_i;

- a monomial x_{vw}^2 recording two copies of the edge $(v; w)$; and
- the two monomials (B_1) and (B_2), where each B_i is a w t_i path in G_i.

The product of the above monomials is equal to the monomial

$$(A_1 \quad (v; w) \quad B_1; A_2 \quad (v; w) \quad B_2):$$

Define the paths $P_i = A_i \quad (v; w) \quad B_i$ for each $i \ 2 \ [2]$.

Since A_i and B_i are paths in G_i, and $(v; w)$ is an edge in both G_1 and G_2, we know that each P_i is an s_i t_i shortest path.

We claim that A_1 does not intersect B_2.

Indeed, suppose to the contrary that A_1 and B_2 intersect at some vertex u. Then P_1 is a shortest path which passes through nodes $u; v; w$ in that order, yet P_2 is a shortest path which passes through $v; w; u$ in that order, which contradicts Proposition 8.25.

Thus A_1 does not intersect B_2 as claimed.

Symmetric reasoning shows that A_2 does not intersect B_1.

Thus the paths P_1 and P_2 have common first intersection at v. Then the pair $hP_1; P_2i$ satisfies the conditions from the claim statement.

Since each pair $hP_1; P_2i$ satisfying the conditions from the claim statement appears as a monomial in eq. (118), and each monomial in eq. (118) is the weight assigned to some such pair, eq. (118) is the enumerating polynomial for the collection of pairs of paths described in the claim statement. This proves the desired result. □

By Claim 8.36, the sum

$$\sum_{v2V} \sum_{w2V_{\text{out}}(v)} \left(S(v)x_{vw}^2\right) R_1(w)R_2(w) \tag{119}$$

enumerates all edge-agreeing pairs whose first intersection v is the beginning of an edge traversed by both paths in the pair. Let F_{start} be the set of such pairs, so that the polynomial from eq. (119) enumerates F_{start}.

We claim that the polynomial from eq. (119) also enumerates F.

To prove this, define $V = F \ n \ F_{\text{start}}$.

Take arbitrary $hP_1; P_2i \ 2 \ V$. Since $hP_1; P_2i$ is edge-agreeing, by Lemma 8.33 this pair is agreeing. Hence P_1 and P_2 have a common first intersection at some vertex v. By Lemma 8.31, these paths also have a common last intersection point at some node w ~~6=~~ v. Since the pair is not in F_{start}, the subpaths $P_1[v; w]$ and $P_2[v; w]$ are distinct.

Now define walks

$$Q_1 = P_1[s_1; v] \quad P_2[v; w] \quad P_1[w; t_1] \quad \text{and} \quad Q_2 = P_2[s_2; v] \quad P_1[v; w] \quad P_2[w; t_2]$$

by swapping the v w subpaths of P_1 and P_2.

By Lemma 8.16 each Q_i is a shortest path, and thus an s_i t_i path in G_i.

Since the P_i are edge-agreeing, and their edge overlap must occur in their v w subpaths, the Q_i are also edge-agreeing. Since $hP_1; P_2i$ is not in F_{start}, vertex v is not the beginning of a common edge traversed by the P_i. Thus, v is also not the beginning of a common edge traversed by the Q_i. This implies that $hQ_1; Q_2i \ 2 \ V$.

Since the w t_i subpaths of each Q_i and P_i are the same, Q_1 and Q_2 have common last intersection at w. So the common first intersect v and common last intersection W are the same for the pairs $hP_1; P_2i$ and $hQ_1; Q_2i$. Moreover, swapping the v w subpaths of paths Q_1 and Q_2 recovers paths P_1 and P_2 respectively.

Consequently, the map sending $hP_1; P_2i$ to node v, the map sending $hP_1; P_2i$ to node w, and the map sending $hP_1; P_2i$ to $hQ_1; Q_2i$ meet the conditions of Lemma 8.17, so the enumerating polynomial for F is the same as the enumerating polynomial for $F \cap V = F_{\text{start}}$.

Since eq. (119) is the enumerating polynomial for F_{start}, by the previous paragraph it enumerates F as well. By the discussion from the first paragraph of this proof, this implies the desired result.

Disagreeing Paths

Let F_{dis} be the family of disagreeing, standard pairs of paths. In this subsection, we show how to compute F_{dis}, the enumerating polynomial for F_{dis}, efficiently.

As with previous enumerations, we compute F_{dis} by following the strategy from Idea 16 to argue that F_{dis} is the enumerating polynomial for a larger class B of pairs of paths containing the family F_{dis}, defined by local constraints around a specified intersection point.

Definition 8.37 (Local Relaxation). For each vertex v, let $\widetilde{B}(v)$ be the set of standard pairs of paths $hP_1; P_2i$ intersecting at v, such that if we let a_i and b_i denote the nodes appearing immediately before and after v on P_i respectively, then

1. $a_1 \neq a_2$,
2. $b_1 \neq b_2$, and
3. $a_1 \neq b_2$.

Let $B(v)$ $\widetilde{B}(v)$ be the collection of pairs in $\widetilde{B}(v)$ such that v is the first vertex in P_1 lying in $P_1 \setminus P_2$. Then define the family

$$B = \bigcup_{v \in V} B(v)$$

by taking the disjoint union of the $B(v)$ collections over all vertices v.

Our high-level approach for computing F_{dis} is as follows:

1. We use the strategy depicted in Figure 6, and for each vertex v, try to compute the polynomial F_v which enumerates all disagreeing, standard pairs of paths $hP_1; P_2i$ such that v is the first vertex in P_1 lying in $P_1 \setminus P_2$. Summing the F_v recovers F_{dis}.
2. We argue that each F_v is the enumerating polynomial for $B(v)$, using Lemma 8.17.
3. We argue that the enumerating polynomials for $B(v)$ and $\widetilde{B}(v)$ are the same for each v, using Lemma 8.17 and additional subpath swapping arguments for undirected graphs.
4. We then enumerate each $\widetilde{B}(v)$ directly, using the conditions from Definition 8.37. By steps 1 through 3 above, this then lets us compute F_{dis}.

This approach can be viewed as another manifestation of Idea 16.

Intuitively, using the subpath swapping idea shown in Figure 7, conditions 1 and 2 from Definition 8.37 (that $a_1 \neq a_2$ and $a_1 \neq b_2$) ensure that v is the first vertex of P_1 lying in $P_1 \setminus P_2$, which helps us argue that $B(v)$ and $\widetilde{B}(v)$ have the same enumerating polynomials. Condition 3 from Definition 8.37 (that $b_1 \neq b_2$) ensures that the paths disagree, which helps us argue that F_{dis} and B have the same enumerating polynomials.

We begin with steps 1 and 2 above, and show that F_{dis} enumerates B.

Lemma 8.38. The enumerating polynomial for B is F_{dis}.

Proof. We start by observing that F_{dis} B.

B Claim 8.39. We have F_{dis} B.

Proof. Take arbitrary $\langle P_1; P_2 \rangle$ 2 F_{dis}.

By definition, $\langle P_1; P_2 \rangle$ is a disagreeing, standard pair of paths. Let v be the first vertex of P_1 lying in $P_1 \setminus P_2$. Let a_i and b_i be the vertices immediately before and after v in P_i for each i 2 $[2]$. From the definition of v, we have $a_1 \neq a_2$ and $a_1 \neq b_2$. We also know that $b_1 \neq b_2$, because if $b_1 = b_2$ then P_1 and P_2 would be edge-agreeing, which by Lemmas 8.27 and 8.33 would contradict the fact that P_1 and P_2 disagree.

Thus $\langle P_1; P_2 \rangle$ satisfies all the conditions from Definition 8.37, so this pair is in B. Since our choice of $\langle P_1; P_2 \rangle$ in F_{dis} was arbitrary, we have F_{dis} B as claimed. □

Our goal is to show that B and F_{dis} have the same enumerating polynomial.

To that end, let $V = B \cap F_{\text{dis}}$ be the collection of agreeing pairs in B.

Take arbitrary $\langle P_1; P_2 \rangle$ 2 V. Let v be the first common intersection of P_1 and P_2 (this vertex exists because P_1 and P_2 agree). Let $w \neq v$ be the last common intersection of the paths P_1 and P_2 (this vertex exists and is distinct from v by Lemma 8.31).

Now define walks

$$Q_1 = P_1[s_1; v] \quad P_2[v; w] \quad P_1[w; t_1] \quad \text{and} \quad Q_2 = P_2[s_2; v] \quad P_1[v; w] \quad P_2[w; t_2]$$

by swapping the v w subpaths in P_1 and P_2.

By Lemma 8.16 each Q_i is a shortest path, and thus an s_i t_i path in G_i.

Let a_i and b_i be the nodes appearing immediately before and after v respectively on P_i, for i 2 $[2]$. Similarly, let a_i' and b_i' be the nodes appearing immediately before and after v respectively on Q_i for i 2 $[2]$.

By the definitions of the Q_i paths, we have $a_1' = a_1$ and $a_2' = a_2$, but $b_1' = b_2$ and $b_2' = b_1$.

Since $\langle P_1; P_2 \rangle$ 2 B, condition 1 of Definition 8.37 implies that $a_1 \neq a_2$. This is equivalent to $a_1' \neq a_2'$. Similarly, condition 2 of Definition 8.37 implies that $b_1 \neq b_2$, which is equivalent to $b_1' \neq b_2'$. Also, since a_1 and b_1 are distinct vertices on the path P_1, we have $a_1 \neq b_1$, which is equivalent to $a_1' \neq b_2'$. Thus $\langle Q_1; Q_2 \rangle$ 2 B.

The s_i v subpaths of P_i and Q_i are the same for each i 2 $[2]$, so Q_1 and Q_2 have common first intersection at v. Thus Q_1 and Q_2 agree, so $\langle Q_1; Q_2 \rangle$ $\not\in$ F_{dis}.

Since $\langle Q_1; Q_2 \rangle$ is in B but not F_{dis}, we have $\langle Q_1; Q_2 \rangle$ 2 V.

Since $b_1 \neq b_2$, the subpaths $P_1[v; w]$ and $P_2[v; w]$ are distinct.

Since the w t_i subpaths of P_i and Q_i are the same for each i 2 [2], Q_1 and Q_2 have common last intersection at w, just like P_1 and P_2. Moreover, swapping the v w subpaths of paths Q_1 and Q_2 recovers paths P_1 and P_2 respectively.

The above discussion implies that the map sending $hP_1; P_2i$ to node v, the map sending $hP_1; P_2i$ to node w, and the map sending $hP_1; P_2i$ to $hQ_1; Q_2i$ meet the conditions of Lemma 8.17, so the enumerating polynomial for B is the same as the enumerating polynomial for $B \cap V = F_{\text{dis}}$. The enumerating polynomial for F_{dis} is F_{dis}, which proves the lemma.

Having established that F_{dis} enumerates B, we move onto step 3 of our approach, and argue that $B(v)$ and $\widetilde{B}(v)$ have the same enumerating polynomial for each vertex v. To show this, we will need a variant of the subpath swapping argument from Lemma 8.17, specialized to undirected graphs, which will allow us to swap subpaths *and reverse their direction*.

Lemma 8.40 (Shortest Path Swapping and Reversing). Let P_1 and P_2 be shortest paths in the weighted, undirected graph G. Let a and b be vertices in $P_1 \setminus P_2$, such that a appears before b on P_1, and b appears before a on P_2. Then the walks obtained by replacing the a b subpath of P_1 with the a b subpath of $\overline{P_2}$, and replacing the b a subpath of P_2 with the b a subpath of $\overline{P_1}$ are shortest paths in G.

Proof. Since P_1 is a shortest path, its a b subpath has length $\text{dist}(a; b)$. Since P_2 is a shortest path, its b a subpath has length $\text{dist}(b; a)$. We have $\text{dist}(a; b) = \text{dist}(b; a)$, because G is undirected, so these subpaths and their reversals have the same length in G. Thus, the walks constructed in the lemma statement by replacing subpaths have the same endpoints and lengths as P_1 and P_2 respectively.

Since G has positive edge weights, any walk in G whose length equals the shortest path distance between its endpoints cannot have repeat vertices. Since P_1 and P_2 are shortest paths, this implies that the new walks are shortest paths as well.

Lemma 8.41 (Vanishing Modulo 2 in Undirected Graphs). Let F be a family of pairs of paths in the undirected graph G, and let V F. Suppose there exist maps $;$ $: V$ V and $: V$ V such that for all $P = hP_1; P_2i$ 2 V,

1. the vertices $a =$ (P) and $b =$ (P) lie in $P_1 \setminus \overline{P_2}$, a appears before b in P_1, b appears before a in P_2, and the subpaths $P_1[a; b]$ and $\overline{P_2}[a; b]$ are distinct;
2. $\overline{(P)} = hQ_1; Q_2i$, where Q_1 is obtained by replacing the a b subpath in P_1 with $\overline{P_2}[a; b]$, and Q_2 is obtained by replacing the b a subpath in P_2 with $\overline{P_1}[a; b]$; and
3. we have $($ $(P)) = P$.

Then the enumerating polynomial for F is the same as the enumerating polynomial for $F \cap V$.

Proof. The proof is nearly identical to the proof of Lemma 8.17.

Let F be the enumerating polynomial for F. By definition, we have

$$F = \sum_{P \in F} (P) = \sum_{P \in F \cap V} (P) + \sum_{P \in V} (P): \tag{120}$$

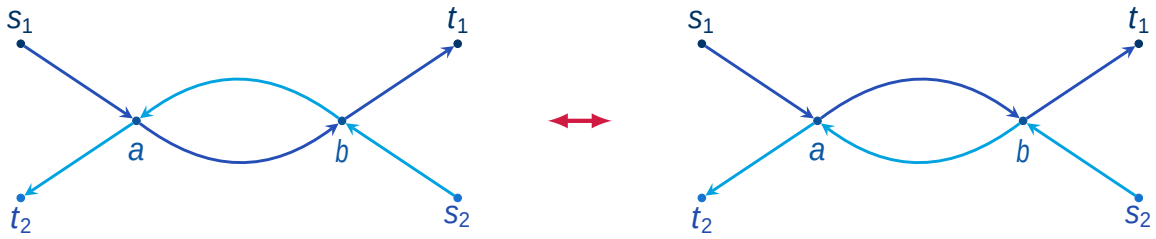

Figure 9: Given paths P_1 and P_2 in an *undirected* graph which intersect at nodes $a = (P_1; P_2)$ and $b = (P_1; P_2)$, such that a appears before b on P_1 and b appears before a on P_2, then if we replace the a b subpath of P_1 with the a b subpath of $\overline{P_2}$ and similarly replace the b a subpath of P_2 with the b a subpath of $\overline{P_1}$ to produce new paths Q_1 and Q_2 respectively, then these pairs $(P_1; P_2) = (Q_1; Q_2)$ have the same weight. Moreover, repeating this subpath replacement transformation on Q_1 and Q_2 recovers the pair $\langle P_1; P_2 \rangle$. This operation is similar to the process from Figure 5. The main difference here is that we use the fact that the graph is undirected, so that edges may be traversed backwards.

Take any $P = \langle P_1; P_2 \rangle \in V$. By property 1 from the lemma statement, the (P) to (P) subpath of P_1 is not equal to the (P) to (P) subpath of $\overline{P_2}$. So by property 2 from the lemma statement, $(P) \notin P$. Consequently, by property 3, we can partition $V = V_1 \sqcup V_2$ into two equally sized pieces such that is a bijection from V_1 to V_2. So we can write

$$\sum_{P \in V} (P) = \sum_{P \in V_1} (P) + \sum_{P \in V_2} (P) = \sum_{P \in V_1} ((P) + ((P))) : \tag{121}$$

By property 2, the multiset of edges traversed by the pair P is the same as the multiset of edges traversed by (P) for all $P \in V$, except some edges may have reversed orientation. Since $x_{uv} = x_{vu}$ for every edge $(u; v)$ in the undirected graph G, we have $(P) = ((P))$ for all $P \in V$.

The subpath swapping is depicted in Figure 9.

Since we work over a field of characteristic two, this implies that

$$\sum_{P \in V_1} ((P) + ((P))) = 0 :$$

Substituting the above equation into eq. (121) implies that

$$\sum_{P \in V} (P) = 0 :$$

Then substituting the above equation into eq. (120) yields

$$F = \sum_{P \in F \cap V} (P) :$$

This proves that F is the enumerating polynomial for $F \cap V$ as desired.

Lemma 8.42. For each vertex v, the enumerating polynomial for $\widetilde{B}(v)$ enumerates $B(v)$.

Proof. Fix a vertex v. By definition, $B(v)$ $\widetilde{B}(v)$.

Our goal is to show that $\widetilde{B}(v)$ and $B(v)$ have the same enumerating polynomial.

To prove this, define $V = \widetilde{B}(v) \setminus B(v)$.

For any pair $P = \langle P_1; P_2 \rangle \in V$, let $u(P)$ denote the first vertex of P_1 lying in $P_1 \setminus P_2$. Since P is not in $B(v)$, we have $u(P) \neq v$.

Let V_{before} be the collection of pairs P in V such that $u(P)$ appears before v in P_2. Similarly, let V_{after} be the collection of pairs P in V such that $u(P)$ appears after v in P_2. Since $u(P)$ must appear before or after v on P_2, we have $V = V_{\text{before}} \cup V_{\text{after}}$. Next, we use subpath swapping arguments to argue that the contributions from V vanish modulo two in the enumerating polynomial for $\widetilde{B}(v)$. We do this in cases, considering the pairs from V_{before} and V_{after} separately.

Case 1: u before v

Take arbitrary $P = \langle P_1; P_2 \rangle \in V_{\text{before}}$. Write $u = u(P)$ for convenience.

By definition, u appears before v in P_2.

Define the walks

$$Q_1 = P_1[s_1; u] \quad P_2[u; v] \quad P_1[v; t_1] \quad \text{and} \quad Q_2 = P_2[s_2; u] \quad P_1[u; v] \quad P_2[v; t_2]$$

obtained by swapping the u v subpaths of P_1 and P_2.

By Lemma 8.16 each Q_i is a shortest path, and thus an s_i t_i path in G_i.

Let a_i and b_i be the nodes appearing immediately before and after v respectively on P_i, for $i \in [2]$. Similarly, let a_i' and b_i' be the nodes appearing immediately before and after v respectively on Q_i for $i \in [2]$.

By the definitions of the Q_i paths, we have $b_1' = b_1$ and $b_2' = b_2$, but $a_1' = a_2$ and $a_2' = a_1$.

Since $P \in \widetilde{B}(v)$, condition 1 of Definition 8.37 implies that $a_1 \neq a_2$, so $a_1' \neq a_2'$. Similarly, condition 2 of Definition 8.37 implies that $b_1 \neq b_2$, so $b_1' \neq b_2'$. Also, since a_2 and b_2 are distinct vertices on the path P_2, we have $a_2 \neq b_2$, so $a_1' \neq b_2'$. Thus $\langle Q_1; Q_2 \rangle \in \widetilde{B}(v)$.

Also, since $u \in Q_1 \setminus Q_2$ appears before v in Q_1, we have $\langle Q_1; Q_2 \rangle \notin B(v)$.

Since $\langle Q_1; Q_2 \rangle$ is in $\widetilde{B}(v)$ but not $B(v)$, we have $\langle Q_1; Q_2 \rangle \in V$. Since vertex u appears before vertex v in Q_2, we in fact have $\langle Q_1; Q_2 \rangle \in V_{\text{before}}$.

Since $a_1 \neq a_2$, the subpaths $P_1[u; v]$ and $P_2[u; v]$ are distinct.

Since the s_i u subpaths of P_i and Q_i are the same for each $i \in [2]$, u is also the first vertex of Q_1 lying in $Q_1 \setminus Q_2$. Moreover, swapping the u v subpaths of Q_1 and Q_2 recovers paths P_1 and P_2 respectively.

The above discussion implies that the map sending P to node $u(P)$, the map sending P to node v, and the map sending P to $\langle Q_1; Q_2 \rangle$ all have domain V_{before} and meet the conditions of Lemma 8.17, so the enumerating polynomial for $\widetilde{B}(v)$ is the same as the enumerating polynomial for $\widetilde{B}(v) \setminus V_{\text{before}} = B(v) \cup V_{\text{after}}$.

So from our arguments in case 1, we have removed the contributions from V_{before} in the enumeration of $\widetilde{B}(v)$. We now continue in case 2, to enumerate the contributions from V_{after}.

Case 2: u after v

Take arbitrary $P = \langle P_1; P_2 \rangle \in V_{\text{after}}$. Write $u = u(P)$ for convenience.

By definition, u appears after v in P_2.

Define the walks

$$Q_1 = P_1[s_1; u] \quad \overline{P_2}[u; v] \quad P_1[v; t_1] \quad \text{and} \quad Q_2 = P_2[s_2; v] \quad \overline{P_1}[v; u] \quad P_2[u; t_2]$$

obtained by replacing the $u \leadsto v$ subpath of P_1 with the $u \leadsto v$ subpath of $\overline{P_2}$, and vice-versa.

By Lemma 8.40 each Q_i is a shortest path, and thus an $s_i \leadsto t_i$ path in G_i.

Let a_i and b_i be the nodes appearing immediately before and after v respectively on P_i, for $i \in [2]$. Similarly, let a_i' and b_i' be the nodes appearing immediately before and after v respectively on Q_i for $i \in [2]$.

By the definitions of the Q_i paths, we have $b_1' = b_1$ and $a_2' = a_2$, but $a_1' = b_2$ and $b_2' = a_1$.

Since a_2 and b_2 are distinct vertices of P_2, we have $a_2 \neq b_2$, so $a_1' \neq a_2'$. Similarly, since a_1 and b_1 are distinct vertices of P_1, we have $a_1 \neq b_1$, so $b_1' \neq b_2'$. Finally, since $P \in \widetilde{B}(v)$, condition 3 of Definition 8.37 implies that $a_1 \neq b_2$, so $a_1' \neq b_2'$. Thus $\langle Q_1, Q_2 \rangle \in \widetilde{B}(v)$.

Also, since $u \in Q_1 \setminus Q_2$ appears before v in Q_1, we have $\langle Q_1, Q_2 \rangle \notin B(v)$.

Since $\langle Q_1, Q_2 \rangle$ is in $\widetilde{B}(v)$ but not $B(v)$, we have $\langle Q_1, Q_2 \rangle \in V$. Since vertex u appears after vertex v in Q_2, we in fact have $\langle Q_1, Q_2 \rangle \in V_{\text{after}}$.

Since $a_1 \neq b_2$, $P_1[u, v]$ and $\overline{P_2}[u, v]$ are distinct paths.

Since the $s_1 \leadsto u$ subpath P_1 and Q_1 are the same, u is also the first vertex of Q_1 lying in $Q_1 \setminus Q_2$. Moreover, replacing the $u \leadsto v$ subpath of Q_1 with the $u \leadsto v$ subpath of $\overline{Q_2}$ and replacing the $v \leadsto u$ subpath of Q_2 with the $v \leadsto u$ subpath of $\overline{Q_1}$, recovers paths P_1 and P_2 respectively.

The above discussion implies that the map sending P to node $u(P)$, the map sending P to node v, and the map sending P to $\langle Q_1, Q_2 \rangle$ all have domain V_{after} and meet the conditions of Lemma 8.41, so the enumerating polynomial for $F = B(v) \cap V_{\text{after}}$ is the same as the enumerating polynomial for $F \cap V_{\text{after}} = B(v)$. Combining this result with the conclusion of case 1, we get that the enumerating polynomial for $\widetilde{B}(v)$ is the same as the enumerating polynomial for $B(v)$, which proves the lemma.

By Lemmas 8.38 and 8.42, we can enumerate F_{dis} simply by enumerating $\widetilde{B}(v)$ for each vertex v. Our next goal is to perform this enumeration efficiently.

To enumerate $\widetilde{B}(v)$, it will be helpful to introduce a polynomial enumerating a target-based analogue of the relaxed source linkages from Definition 8.18.

Definition 8.43 (Relaxed Target Linkages)**.** Given a vertex v, let $\widetilde{T}(v)$ be the collection of pairs of paths $\langle P_1, P_2 \rangle$, where each P_i is a $v \leadsto t_i$ path in G_i, and the second vertices of P_1 and P_2 are distinct. Let $T(v)$ be the enumerating polynomial for $\widetilde{T}(v)$.

Lemma 8.44. For each vertex v, we have

$$T(v) = R_1(v)R_2(v) \sum_{w \in V_{\text{out}}(v)} x_{vw}^2 R_1(w)R_2(w).$$

Proof. This follows from symmetric reasoning to the proof of Lemma 8.20.

For any fixed vertex v, by Lemma 8.19 the product $S(v)T(v)$ should enumerate pairs of paths satisfying conditions 1 and 2 from Definition 8.37. To enumerate $\widetilde{B}(v)$, we want to additionally enforce condition 3 from Definition 8.37. Said another way, we want to subtract off the pairs of paths which *violate* condition 3. We will do this using the following polynomial, that enumerates pairs of paths which do not satisfy condition 3 from Definition 8.37.

Definition 8.45. Given a vertex v, let $M(v)$ be the collection of standard pairs of paths $\langle P_1; P_2 \rangle$ intersecting at v, such that the vertex immediately before v on P_1 is the same as the vertex immediately after v on P_2. Let $M(v)$ be the enumerating polynomial for $M(v)$.

Recall the definition of the mixed-neighborhood $V_{\mathsf{mix}}(v)$ of a vertex v from eq. (113). From Definition 8.45, we see that for any pair of paths in $M(v)$, the vertex immediately before v in P_1 must belong to $V_{\mathsf{mix}}(v)$. This motivates the following formula for $M(v)$.

Lemma 8.46. For each vertex v, we have

$$M(v) = \sum_{u \in V_{\mathsf{mix}}(v)} L_1(u) x_{uv} R_1(v) L_2(v) x_{vu} R_2(u).$$

Proof. For any vertices u and v, define $F(u; v)$ to be the collection of standard pairs of paths $\langle P_1; P_2 \rangle$ such that P_1 traverses edge $(u; v)$ and P_2 traverses edge $(v; u)$.

To prove the lemma, we first establish the following claim.

Claim 8.47. For any vertices u and v with $u \in V_{\mathsf{mix}}(v)$, the polynomial

$$L_1(u) x_{uv} R_1(v) L_2(v) x_{vu} R_2(u) \tag{122}$$

enumerates $F(u; v)$.

Proof. Take arbitrary $\langle P_1; P_2 \rangle \in F(u; v)$. Then we can decompose the paths as

$$P_1 = P_1[s_1; u] \quad (u; v) \quad P_1[v; t_1] \quad \text{and} \quad P_2 = P_2[s_2; v] \quad (v; u) \quad P_2[u; t_2].$$

Since path $P_1[s_1; u]$ is enumerated in $L_1(u)$, edge $(u; v)$ is enumerated by x_{uv}, path $P_1[v; t_1]$ is enumerated by $R_1(v)$, path $P_2[s_2; v]$ is enumerated by $L_2(v)$, edge $(v; u)$ is enumerated by x_{vu}, and path $P_2[u; t_2]$ is enumerated by $R_2(u)$, the expansion of the polynomial from eq. (122) contains $(P_1; P_2)$ as a monomial.

Conversely, any monomial in the expansion of eq. (122) is equal to the product of

- a monomial (A_1) from $L_1(u)$, where A_1 is an $s_1 \quad u$ path in G_1;
- a monomial x_{uv} recording the edge $(u; v)$;
- a monomial (B_1) from $R_1(v)$, where B_1 is a $v \quad t_1$ path in G_1;
- a monomial (A_2) from $L_2(u)$, where A_2 is an $s_2 \quad v$ path in G_2;
- a monomial x_{vu} recording the edge $(v; u)$; and
- a monomial (B_2) from $R_2(v)$, where B_2 is a $u \quad t_2$ path in G_2.

The product of the above monomials is equal to the monomial

$$(A_1 \quad (u; v) \quad B_1; A_2 \quad (v; u) \quad B_2).$$

Define the paths

$$P_1 = A_1 \quad (u; v) \quad B_1 \quad \text{and} \quad P_2 = A_2 \quad (v; u) \quad B_2.$$

Since $u \in V_{\text{mix}}(v)$, (u, v) is an edge in G_1, and (v, u) is an edge in G_2.
Consequently, P_i is an $s_i \to t_i$ path in G_i for each $i \in [2]$.
Since P_1 traverses (u, v) and P_2 traverses (v, u), we have $\langle P_1, P_2 \rangle \in F(u, v)$.
Since every pair in $F(u, v)$ contributes a monomial to eq. (122), and every monomial of the polynomial eq. (122) is the weight of a pair in $F(u, v)$, the desired result holds. □

For any pair of paths $\langle P_1, P_2 \rangle \in M(v)$, there exists a unique vertex u such that u appears immediately before v on P_1 and immediately after v on P_2. Since P_i is a path in G_i for $i \in [2]$, any such vertex u must lie in $V_{\text{in}}^1(v) \setminus V_{\text{out}}^2(v) = V_{\text{mix}}(v)$ by definition.
Then by applying Claim 8.47 to each $u \in V_{\text{mix}}(v)$, we get that

$$\sum_{u \in V_{\text{mix}}(v)} L_1(u) x_{uv} R_1(v) L_2(v) x_{vu} R_2(u)$$

is the enumerating polynomial for $M(v)$, which proves the desired result.

We now present our formula for F_{dis}, in terms of the polynomials $S(v)$ and $T(v)$ defined in Definitions 8.15 and 8.43 respectively. This formula comes from implementing the strategy discussed in the paragraph preceding Definition 8.45.

Lemma 8.48 (Enumerating Disagreeing Pairs)**.** We have

$$F_{\text{dis}} = \sum_{v \in V} (S(v) T(v) \setminus M(v)).$$

Proof. We prove the lemma by using the following claim.

Claim 8.49. For any vertex v, the polynomial

$$S(v) T(v)$$

enumerates all standard pairs of paths $\langle P_1, P_2 \rangle$ intersecting at v, such that the nodes immediately before v on paths P_1 and P_2 are distinct, and the nodes immediately after v on P_1 and P_2 are distinct.

Proof. Fix a vertex v. Let $\langle P_1, P_2 \rangle$ be any standard pair of paths intersecting at v, such that the nodes immediately before v on paths P_1 and P_2 are distinct, and the nodes immediately after v on P_1 and P_2 are also distinct. Then we can write

$$P_i = A_i \cdot B_i$$

where A_i is an $s_i \to v$ path in G_i, and B_i is a $v \to t_i$ path in G_i, for each $i \in [2]$, such that the penultimate vertices of A_1 and A_2 are distinct, and the second vertices of B_1 and B_2 are distinct. Then by definition $S(v)$ includes the monomial (A_1, A_2) and $T(v)$ includes the monomial (B_1, B_2), so their product $S(v)T(v)$ includes the monomial

$$(A_1, A_2) \cdot (B_1, B_2) = (P_1, P_2).$$

Conversely, any monomial in the expansion of $S(v)T(v)$ is of the form

$$(A_1; A_2) \quad (B_1; B_2)$$

where the A_i are $s_i \quad v$ paths in G_i with distinct penultimate nodes, and the B_i are $v \quad t_i$ paths in G_i with distinct second nodes. Then if we set $P_i = A_i \quad B_i$ for each $i \in [2]$, $\langle P_1; P_2 \rangle$ is a standard pair of paths intersecting at v, such that P_1 and P_2 have with distinct nodes immediately before v, and distinct nodes immediately after v.

This proves the claim. □

We can now prove the following.

Claim 8.50. For each vertex v,

$$S(v)T(v) \quad M(v)$$

is the enumerating polynomial for $\widetilde{B}(v)$.

Proof. Fix a vertex v. By Claim 8.49, $S(v)T(v)$ enumerates all standard pairs of paths $\langle P_1; P_2 \rangle$ intersecting at v, such that if we let a_i and b_i denote the nodes appearing immediately before and after v on P_i respectively, we have $a_1 \not\sim a_2$ and $b_1 \not\sim b_2$.

By Definition 8.45, $M(v)$ enumerates all standard pairs of paths $\langle P_1; P_2 \rangle$ intersecting at vertex v, such that (using the a_i and b_i notation from above) $a_1 = b_2$. Since P_1 and P_2 are paths, they do not have repeat vertices. In particular, $a_1 \not\sim b_1$ and $a_2 \not\sim b_2$. Since $a_1 = b_2$, this implies that $a_1 \not\sim a_2$ and $b_1 \not\sim b_2$.

So we can equivalently state that $M(v)$ is the enumerating polynomial for all standard pairs of paths $\langle P_1; P_2 \rangle$ intersecting at v, such that $a_1 \not\sim a_2$, $b_1 \not\sim b_2$, and $a_1 = b_2$.

Since every pair of paths $\langle P_1; P_2 \rangle$ must have either $a_1 = b_2$ or $a_1 \not\sim b_2$, we get that

$$S(v)T(v) \quad M(v)$$

is the enumerating polynomial for all standard pairs of paths $\langle P_1; P_2 \rangle$ intersecting at v, such that $a_1 \not\sim a_2$, $b_1 \not\sim b_2$, and $a_1 \not\sim b_2$. By Definition 8.37, this proves the desired result. □

By Claim 8.50 and Lemma 8.42, for each vertex v, the polynomial

$$S(v)T(v) \quad M(v)$$

enumerates $B(v)$.

Since B is the disjoint union of the $B(v)$ sets over all vertices v, we get that

$$\sum_{v \in V} (S(v)T(v) \quad M(v))$$

is the enumerating polynomial for B.

By Lemma 8.38, this means that the above polynomial enumerates F_{dis}.

Since F_{dis} is the enumerating polynomial for F_{dis}, this proves the desired result.

Having presented formulas for F_{agree} and F_{dis}, we present our algorithm for 2-DSP in weighted undirected graphs in Algorithm 10.

Note that steps 1 to 3 of Algorithm 10 are the same as steps 1 to 3 of Algorithm 9, and step 9 of Algorithm 10 is the same as step 6 of Algorithm 9, because our algorithms for 2-DSP in DAGs and undirected graphs have the same overall structure. Just like in Algorithm 9, we never compute polynomials explicitly in Algorithm 10, and instead just compute polynomial evaluations with respect to the random assignment in step 2 of Algorithm 10.

Lemma 8.51 (Undirected Algorithm Correctness). Algorithm 10 solves 2-DSP in weighted undirected graphs with high probability.

Proof. By Lemmas 8.20 and 8.44, step 4 of Algorithm 10 correctly computes $S(v)$ and $T(v)$ for each vertex v.

By Lemma 8.46, step 5 of Algorithm 10 correctly computes $M(v)$ for each vertex v.

By Lemma 8.35, step 6 of Algorithm 10 correctly computes F_{agree}.

By Lemma 8.48, step 7 of Algorithm 10 correctly computes F_{dis}.

By Lemma 8.30, step 8 of Algorithm 10 correctly computes $F_{\setminus}$.

By Proposition 8.9, F_{disj} is a nonzero polynomial if and only if G contains vertex-disjoint s_i t_i shortest paths for $i \in [2]$. By definition, F_{disj} is a polynomial of degree at most $2n$. Then by Proposition 6.1 and our choice of q in eq. (111), with high probability the evaluation of F_{disj} on the random assignment from step 2 of Algorithm 10 is nonzero if and only if G contains a solution to the 2-DSP problem. Thus with high probability, Algorithm 10 returns the correct answer to the 2-DSP problem in step 9.

Proof of Theorem 8.1. By Lemma 8.24, Algorithm 10 solves the 2-DSP problem in weighted undirected graphs. It remains to show that Algorithm 10 runs in linear time.

Step 1 of Algorithm 10 takes linear time by Proposition 8.7.

Step 2 of Algorithm 10 takes linear time because we spend $O(1)$ time at each edge of G.

Step 3 of Algorithm 10 takes linear time by Corollary 8.12.

For each fixed vertex v, computing $S(v)$ and $T(v)$ using the formulas in step 4 of Algorithm 10 takes $O(\deg_{\mathsf{in}}(v))$ and $O(\deg_{\mathsf{out}}(v))$ time respectively. Summing this runtime bound over all vertices v, we see that step 4 of Algorithm 10 takes $O(m)$ time.

For each fixed vertex v, computing $M(v)$ using the formula in step 5 of Algorithm 10 takes $O(\deg_{\mathsf{in}}(v))$ time, since $V_{\mathsf{mix}}(v)$ contains at most $\deg_{\mathsf{in}}(v)$ nodes. Summing this runtime bound over all vertices v, we see that step 5 of Algorithm 10 takes $O(m)$ time.

The sum from the formula in step 6 of Algorithm 10 has a summand for each pair of vertices $(v; w)$ with $w \in V_{\mathsf{out}}(v)$. Each such pair $(v; w)$ must be an edge in the graph, so the sum has at most m terms. Step 6 of Algorithm 10 then takes $O(m)$ time, since we add and multiply $O(m)$ field elements.

Step 7 of Algorithm 10 takes $O(n)$ time because we add and multiply $O(n)$ field elements.

Step 8 of Algorithm 10 takes $O(1)$ time given our previous computations.

Step 9 of Algorithm 10 also takes $O(1)$ time given our previous computations.

So overall Algorithm 10 runs in linear time as claimed.

Algorithm 10. The 2-DSP Algorithm in Undirected Graphs

Inputs: An undirected graph G, with specified sources $s_1; s_2$ and targets $t_1; t_2$.
Returns: YES if G contains disjoint s_i t_i shortest paths for i 2 [2], NO otherwise.

1. For each i 2 [2], compute the s_i-shortest paths DAG G_i.
2. Sample independent, uniform random values x_{uv} from F for each $(u; v)$ 2 E.
3. For every vertex v and each i 2 [2], compute $L_i(v)$ and $R_i(v)$.
4. For every vertex v, compute

$$S(v) = L_1(v)L_2(v) \quad \sum_{u2V_{\text{in}}(v)} L_1(u)L_2(u)x_{uv}^2$$

   and

$$T(v) = R_1(v)R_2(v) \quad \sum_{w2V_{\text{out}}(v)} x_{vw}^2 R_1(w)R_2(w):$$

5. For each vertex v, compute

$$M(v) = \sum_{u2V_{\text{mix}}(v)} L_1(u)x_{uv}R_1(v)L_2(v)x_{vu}R_2(u):$$

6. Compute

$$F_{\text{agree}} = \sum_{v2V} \sum_{w2V_{\text{out}}(v)} \left(S(v)x_{vw}^2\right) R_1(w)R_2(w):$$

7. Compute

$$F_{\text{dis}} = \sum_{v2V} (S(v)T(v) \quad M(v)):$$

8. Compute

$$F_{\setminus} = F_{\text{agree}} + F_{\text{dis}}:$$

9. Compute

$$F_{\text{disj}} = L_1(t_1)L_2(t_2) \quad F_{\setminus}:$$

   Return YES if F_{disj} is nonzero, NO if F_{disj} is zero.

8.6 Additional Consequences

Finding Disjoint Shortest Paths

Our algorithms for 2-DSP in weighted DAGs and undirected graphs detect if G contains vertex-disjoint s_i t_i shortest paths, but do not explicitly return these paths if they exist. How can we solve the search problem of *finding* disjoint shortest paths, when they exist?

A natural approach is to run our detection algorithms for 2-DSP multiple times on subgraphs of G, to identify the edges which belong in a solution.

For example, we could start by solving 2-DSP on G. If the answer is YES, then, for each edge e 2 $V^1_{\text{out}}(s_1)$, we can solve 2-DSP in the graph obtained by taking G and contracting the edge e. The answer to 2-DSP is YES on such a graph precisely when G contains vertex-disjoint s_i t_i such that e is the first edge on the s_1 t_1 path. So one of these calls will return YES, and help us identify the first edge e on a solution path. We can then repeat this strategy, continuing to contract edges until we find all edges on an s_1 t_1 shortest path belong to a solution. We then delete all nodes on this path, and look for an s_2 t_2 path in the resulting graph to find the desired solution paths.

The downside of this approach is that in the worst case, we might end up solving up to m instances of 2-DSP, as we try deleting each edge in G. This then would lead to an algorithm taking (m^2) time, which is far slower than we would hope for.

We can get a slightly better algorithm, by exploiting the fact that our 2-DSP algorithms involve evaluating a disjoint paths *polynomial*.

An *arithmetic circuit* C is a list of steps for building up a polynomial P (intuitively a polynomial version of a Boolean circuit, which is defined in Definition 4.20). Formally, the circuit can be viewed as a sequence of gates. Each variable of P corresponds to one of initial gates of C. Each later gate is either the product of two earlier gates, an F-linear combination of two earlier gates, the product of a field element with a previous gate, or the sum of a field element with a previous gate. In this way, each gate computes a polynomial, defined inductively in terms of the polynomials computed by earlier gates. The circuit C computes a polynomial P if its final gate computes P. The size jCj of circuit C is the number of gates it contains. Given an arithmetic circuit C computing a polynomial P, we can evaluate P at a point over F in jCj time, just scanning through C and computing the evaluation of each gate at the given point.

Since Algorithms 9 and 10 build up F_{disj} using only polynomial addition and multiplication, Lemmas 8.24 and 8.51 and the proofs of Theorems 8.1 and 8.2 show that these algorithms actually describe linear-size arithmetic circuits for F_{disj} over weighted DAGs and undirected graphs (i.e., if we ignore the random evaluation from step 2 of Algorithms 9 and 10, these algorithms yield descriptions of arithmetic circuits for F_{disj}).

Proposition 8.52. Over weighted DAGs and undirected graphs, F_{disj} admits an arithmetic circuit of size $O(m)$.

We recall that given a variable x and a polynomial P, the polynomial $(@=@x)P$ denotes the partial derivative of P with respect to x. The following result on partial derivatives will help us use Proposition 8.52 to find disjoint shortest paths.

Proposition 8.53 (Baur-Strassen Theorem). Given an arithmetic circuit C of size s computing a polynomial P, we can compute a multi-output arithmetic circuit C' of size $O(s)$, which computes the polynomial $(@=@x)P$ for every variable x of P.

Proposition 8.53 was originally proved in [BS83]. For a modern exposition of the proof of Proposition 8.53, we refer the reader to [SY09, Theorem 2.5].

Proof of Theorem 8.3. Set q to be the smallest positive integer with

$$2^q \quad 2m^2n^2:$$

We work over $\mathsf{F} = \mathsf{F}_{2^q}$. Note that $q = \quad (\log n)$, so addition and multiplication over F takes $O(1)$ time over the Word RAM model. For this proof we use the value of q defined above, instead of the value of q from Equation (111).

Given an edge $(u; v)$, the polynomial $(@=@x_{uv})\,F_{\text{disj}}$ is nonzero if and only if edge $(u; v)$ appears in some solution to the 2-DSP problem. Also, $(@=@x_{uv})F_{\text{disj}}$ has degree less than $2n$. So by Proposition 6.1 and our choice of field size in eq. (111), $(@=@x_{uv})F_{\text{disj}}$ has nonzero evaluation at a uniform random assignment over F if and only if $(u; v)$ is an edge occurring in some pair of disjoint shortest paths, for fixed $(u; v)$ with probability at least $1 \quad 1=(m^2n)$. By taking a union bound over all edges x_{uv} in the graph, this holds for all edges $(u; v)$ with probability at least $1 \quad 1=(mn)$.

By Propositions 8.52 and 8.53, we can compute all first-order partial derivatives of F_{disj} at a given random evaluation point in $O(m)$ time. We pick an edge $(s_1; v)$ such that $(@=@x_{s_1v})F_{\text{disj}}$ has nonzero evaluation. Then we delete vertex s_1 from G, and consider a smaller instance of 2-DSP on the graph, where source s_1 is replaced with v. We can repeat this process on the new instance, to find the first edge on a $v \quad t_1$ shortest path which is disjoint from some $s_2 \quad t_2$ shortest path. Repeating this process at most n times, we can recover an $s_1 \quad t_1$ shortest path P_1, which is disjoint from some $s_2 \quad t_2$ shortest path.

At this point, we just delete all vertices of P_1 from the original graph G, find an $s_2 \quad t_2$ shortest path P_2 in the resulting graph in linear time, and then return $hP_1; P_2i$ as our answer.

Overall, we compute at most n evaluations of arithmetic circuits of size $O(m)$, so the algorithm runs in $O(mn)$ time. Moreover, by a union bound, all the random evaluations correctly detect disjoint shortest paths when they exist with probability at least $1 \quad 1=m$, so the algorithm is correct with high probability.

Edge-Disjoint Paths

We defined k-DSP in terms *vertex-disjoint* paths. We can of course also ask questions about edge-disjoint paths. In some contexts, this might even be more natural to study (for example, in Chapter 7 we viewed connectivity in terms of edge-disjoint paths as the primary definition, and its vertex-disjoint analogue of vertex connectivity as a secondary concept).

> k-Edge Disjoint Shortest Paths (k-EDSP)
>
> Given a graph G with specified nodes $s_1; : : : ; s_k$ and $t_1; : : : ; t_k$, determine if G contains edge-disjoint $s_i \quad t_i$ shortest paths.

For any constant k, there is a simple reduction from k-EDSP on n nodes and m edges to k-DSP on $O(m + n)$ nodes and $O(m)$ edges. We can use this reduction together with Theorems 8.1 and 8.2 to derive linear-time algorithms for 2-EDSP in weighted DAGs and undirected graphs.

Proposition 8.54 (Edge-Disjoint Vertex-Disjoint)**.** There is an $O(k(m + n))$ time reduction from k-EDSP on n vertices and m edges to k-DSP on $m + k(n + 2)$ nodes and $2k(m + 1)$ edges.

Proof. Let $G = (V; E)$ be an arbitrary instance of k-EDSP on n vertices and m edges, with sources $s_1; : : : ; s_k$ and targets $t_1; : : : ; t_k$.

We construct a graph G', an instance of k-DSP, as follows. For every vertex $v \in V$, G' has k nodes $v_1; : : : ; v_k$. We call the v_i the *copies* of v in G'. For every edge $e \in E$, G' has a node e. For every edge $e = (v; w) \in E$, we include edges in G' from v_i to e and from e to w_i for all $i \in [k]$. If $e = (v; w)$ had weight $\ell(v; w)$ in G, then the $(v_i; e)$ and $(e; w_i)$ edges in G each have weight $\ell(v; w)$. We also introduce new sources $s'_1; : : : ; s'_k$ and targets $t'_1; : : : ; t'_k$ in G'. For all $i; j \in [k]$, we add edges from s'_i to $(s_i)_j$ and from $(t_i)_j$ to t'_i of weight 1.

By definition, G' has $m + k(n + 2)$ nodes and $2k(m + 1)$ edges.

Moreover, we can construct G' in $O(k(m + n))$ time, given G.

We now prove that solving k-DSP on G' corresponds to solving k-EDSP on G.

Suppose we have vertex-disjoint s'_i t'_i shortest paths P'_i in G'. Since each P'_i is a shortest path, we know it never traverses two copies of the same vertex $v \in V$ (if it did, we could remove the subpath between two consecutive copies of v that P'_i enters to obtain a shorter s'_i t'_i path). We map each P'_i to an s_i t_i shortest path P_i in G, by having P_i pass through the vertices $v \in V$ for which P'_i contains a copy of v, in the order the copies appear in P'_i.

By construction, P_i has length r if and only if P'_i has length $2r + 2$. Since the P'_i are shortest paths, this implies that the P_i are also shortest paths. The P_i are also edge-disjoint, since if some P_i and P_j overlap at an edge e, the paths P'_i and P'_j would overlap at node e in G', which would contradict the assumption that the P'_i are vertex-disjoint.

Thus, any solution to k-DSP on G' pulls back to a solution to k-EDSP on G.

Conversely, given edge-disjoint s_i t_i shortest paths P_i in G of the form

$$P_i = \langle v_{i;1}; : : : ; v_{i;r_i} \rangle$$

we can produce s'_i t'_i paths P'_i in G' of the form

$$P'_i = \langle s'_i; (v_{i;1})_i; ((v_{i;1})_i; (v_{i;2})_i); (v_{i;2})_i; : : : ; (v_{i;r_i})_i; t'_i \rangle;$$

By construction, P'_i has length $2r + 2$ if and only if P'_i has length r. So since the P_i are shortest paths, the P'_i are shortest paths as well. These paths are vertex-disjoint because each P'_i only uses the i^{th} copies of $v \in V$, and the P_i were edge-disjoint, so the P'_i cannot overlap at any nodes of the form $e \in E$.

This proves the desired result.

Corollary 8.55. There are algorithms solving 2-EDSP on weighted DAGs and undirected graphs in linear time.

Proof. By setting $k = 2$ in Proposition 8.54, we can reduce 2-EDSP on graphs with n vertices and m edges to 2-DSP on graphs with $O(m+n)$ vertices and $O(m)$ edges. Running the linear-time algorithms of Theorems 8.1 and 8.2 on the resulting 2-DSP instance solves the original 2-EDSP instance in linear time as well.

8.7 Open Problems

Our linear-time algorithms for 2-DSP are randomized, and only detect the existence of disjoint shortest paths, instead of returning those paths when they exist. These are common limitations for algorithms which employ the algebraic framework discussed in Chapter 6. It is an interesting research direction to design algorithms for 2-DSP which overcome these limitations, yet still run in linear time.

Open Problem 38. Can 2-DSP be solved in *deterministic* linear time over weighted DAGs? What about over undirected graphs?

Open Problem 39. Is there a linear-time algorithm which solves 2-DSP over weighted DAGs and *returns a pair of solution paths* when such a pair exists? Is this possible over undirected graphs?

It is also natural to ask whether we can solve 2-DSP over *general directed graphs* in linear time. The only known polynomial-time algorithm for 2-DSP over general directed graphs is presented in [BK17, Theorem 2]. Their exposition actually solves the more general 2-EDSP problem, and their approach necessarily takes (m^2) time, because they construct a larger graph, whose nodes are pairs of edges in the original graph. To solve 2-DSP instead of 2-EDSP, it seems likely that one could modify the approach of [BK17] to only work with a graph whose nodes are pairs of vertices (rather than edges) in the original graph. Such an approach would still take (n^2) time however.

It is unclear if the algebraic approach employed in our 2-DSP algorithms on DAGs and undirected graphs can help design linear-time algorithms for 2-DSP in general directed graphs. The main issues are that in directed graphs, shortest paths can intersect in more complicated ways (e.g., Proposition 8.25 and Lemma 8.27 do not necessarily hold), and we cannot perform subpath swaps which reverse the orientations of edges (so that we cannot apply Lemma 8.41 directly). On the other hand, [BK17] solves 2-DSP on directed graphs by reducing the problem to several instances of 2-DP in DAGs. So perhaps combining the ideas from this chapter with the method of [BK17] could be fruitful.

Open Problem 40. Can 2-DSP be solved over directed graphs in linear time? Or does a plausible hardness hypothesis rule out such an algorithm?

It would also be interesting to design faster algorithms for k-DSP when k 3. The simplest interesting case to consider is 3-DSP over DAGs, where the current fastest algorithm

runs in $O(mn^2)$ time. This is also the fastest known algorithm for 3-DP over DAGs. In dense graphs with $m = \quad (n^2)$ this runtime becomes $O(n^4)$, so obtaining a truly subquartic time algorithm for this problem on general DAGs would be very interesting.

Open Problem 41. Can 3-DP or 3-DSP be solved over DAGs in $O(n^4 \ ")$ time for some constant $" > 0$? Or is such a runtime ruled out by some plausible hardness hypothesis?

It is not clear if the algebraic techniques we use to solve 2-DSP can help solve 3-DSP in DAGs as well. In our proofs, we used subpaths swapping arguments to show that enumerating polynomials for pairs of shortest paths in DAGs simplify greatly modulo two. In particular, as mentioned in Idea 15, when enumerating pairs of shortest paths in a DAG modulo two, it generally suffices to restrict our attention to pairs of paths whose intersections form single common subpath. This is intuitively because we can use Lemma 8.17 to argue that contributions from pairs of paths whose intersections do not lie on a common subpath vanish modulo two. To solve 3-DSP in DAGs efficiently using these algebraic techniques, we would ideally want to argue that *triples* of shortest paths in DAGs simplify in an analogous fashion modulo two.

Unfortunately, when enumerating triples of shortest paths in DAGs modulo two, restricting our attention to the subcollection of triples where the intersections for each pair of paths in the triple form a common subpath does not suffice. This is because there exist DAGs with an odd number of triples of shortest paths between specified terminal pairs, with the property that each triple contains a pair of paths which do not just overlap at a single common segment. For example, such a DAG is pictured in Figure 10. This means that polynomials which enumerate triples of shortest paths in DAGs do not simplify as much modulo two as polynomials which just enumerate *pairs* of shortest paths, so applying algebraic methods to solve 3-DSP faster may prove difficult.

Designing faster algorithms for k-DSP for $k \quad 3$ over undirected graphs would also be very interesting. As discussed in Section 8.1, the current best algorithms for k-DSP in undirected graphs run in $n^{O(k\,k!)}$ time, and the current fastest algorithm for 3-DSP in undirected graphs takes $O(n^{292})$ time [BNRZ21]. Obtaining more practical runtimes for this problem for small values of k would mark significant progress in our understanding of the structure of shortest paths in graphs.

Open Problem 42. Is there an algorithm solving 3-DSP over undirected graphs, with a *practical* polynomial runtime?

Open Problem 43. Can k-DSP be solved over undirected graphs in $n^{o(k\,k!)}$ time?

Even though existing algorithms for k-DSP over undirected graphs are significantly slower than the fastest algorithms we have for k-DSP over DAGs, the best conditional lower bounds we have for the time complexities of these problems are essentially the same. These lower bounds come via reductions from the k-Clique problem, defined below.

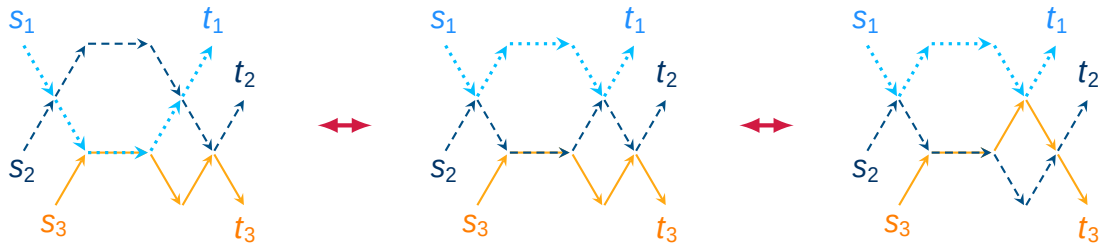

Figure 10: An example of an unweighted DAG and its three triples of s_i t_i shortest paths for i 2 [3]. We can move between the triples on the left and the center by swapping subpaths between the s_1 t_1 and s_2 t_2 paths. We can move between the triples on the center and the right by swapping subpaths between the s_2 t_2 and s_3 t_3 subpaths. The pictured triples account for all collections of paths which can be generated by starting from these triples and applying subpath swapping operations. Since each triple has the same monomial weight and there are three of them, the contributions from these triples do not vanish when enumerating modulo two. Yet, each triple contains a pair of paths whose intersections do not form a common subpath.

k-Clique

Given an undirected graph G with vertex set V_1 t t V_k, where $jV_ij = n$ for each i 2 [k], determine if there exist vertices $v_1; : : : ; v_k$ with v_i 2 V_i for all i 2 [k], such that $(v_i; v_j)$ is an edge of G for all distinct indices $i; j$ 2 [k].

Under the Exponential Time Hypothesis (ETH) (a popular hardness hypothesis in complexity, whose definition can be found, for example, in [CFK+16, Section 14.1]) the k-Clique problem requires $n^{(k)}$ time to solve [CFK+16, Theorem 14.21]. There are $O((kn)^2)$ time reductions from k-Clique to k-DSP on unweighted DAGs [AWW24, Theorem 7] and undirected graphs [BFG24, Proof of Theorem 1] over $O((kn)^2)$ vertices, so under ETH these k-DSP requires $n^{(k)}$ time to solve as well, over DAGs and undirected graphs. Since k-DSP over DAGs can be solved in $O(mn^{k\ 1})$ time, this shows that under ETH, $n^{(k)}$ is the correct runtime for k-DSP over DAGs. However, it remains unclear what the best runtime for k-DSP over undirected graphs should be —is the truth near the current upper bound of $n^{O(k\ k!)}$, or is it closer to the current ETH-based lower bound of $n^{(k)}$?

The difference between the (k) and $O(k\ k!)$ factors in the exponent for the current lower and upper bounds for the runtime of k-DSP in undirected graphs is quite large, and we view it as an important open problem to close this gap. In particular, it would be interesting to establish a conditional lower bound which separates the complexities of k-DSP in undirected graphs and DAGs. On the other hand, if designing such a lower bound proves difficult, that might suggest that far faster algorithms for k-DSP in undirected graphs may exist.

Open Problem 44. Does k-DSP over weighted, undirected graphs require $n^{!(k)}$ time to solve under some plausible hardness hypothesis?

The best algorithms we have for k-DSP over weighted DAGs run as quickly as the best algorithms we have for k-DP over DAGs, even though the former is a generalization of the

latter [AWW24, Proposition 54]. Yet, the current best conditional lower bounds for k-DP on DAGs remain worse than the best lower bounds we have for k-DSP on DAGs (see [Chi23], and the discussion between Corollary 8 and Theorem 9 of [AWW24]). In this context, resolving the gap between these lower bounds, or getting faster algorithms for k-DP when $k \geq 3$ would be very interesting.

Open Problem 45. Can k-DP on DAGs be solved faster than k-DSP on weighted DAGs for some constant k? Or, is k-DP on DAGs at least hard as k-DSP on weighted DAGs for every k, under some plausible hardness hypothesis?

We introduced k-DSP as an optimization variant of k-DP. Another natural attempt at turning k-DP into an optimization problem produces the following task:

> **MinSum k-Disjoint Paths** (MinSum k-DP)
>
> Given a graph G with specified nodes $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$, determine the smallest length ℓ for which G contains internally vertex-disjoint $s_i \rightsquigarrow t_i$ paths P_i, such that the sum of the lengths of the P_i is at most ℓ, or report that no such positive integer ℓ exists.

Unlike in k-DSP, in MinSum k-DP we do not require the individual solution paths to be shortest paths, but instead just want to detect the existence of disjoint paths connecting specified terminal pairs with minimum total length.

For all $k \geq 2$, this problem is NP-hard on general directed graphs since solving the easier problem of 2-DP on directed graphs is NP-hard [FHW80]. So from the view of polynomial-time algorithms, MinSum k-DP is only interesting on restricted classes of graphs.

It is known that MinSum 2-DP can be solved in $O(n^{3+\omega})$ time over unweighted, undirected graphs [BHK22, Section 6]. No polynomial-time algorithms or hardness results appear to currently be known for MinSum 2-DP over weighted undirected graphs, or MinSum k-DP over undirected graphs for any constant $k \geq 3$.

Open Problem 46. Can MinSum 2-DP be solved over unweighted undirected graphs in faster than $O(n^{3+\omega})$ time? Does some plausible hardness hypothesis rule out the possibility of solving this problem in *linear time*?

Open Problem 47. Is MinSum 2-DP polynomial-time solvable over *weighted* undirected graphs? Or is this problem NP-hard?

Open Problem 48. Is MinSum 3-DP polynomial-time solvable over undirected graphs? Or is this problem NP-hard?

Since the known polynomial-time algorithms for MinSum 2-DP use algebraic techniques, they are randomized (just like our 2-DSP algorithms). It would be interesting to remove the use of randomness in this approach.

Open Problem 49. Can MinSum 2-DP be solved over unweighted undirected graphs in *deterministic* polynomial time?

Given the utility of algorithms for k-DP over DAGs in helping solve k-DSP over undirected graphs, to make progress on Open Problems 46 to 48 it may prove useful to first try and design efficient algorithms for MinSum k-DP over DAGs. We are not aware of previous work studying this problem.

Open Problem 50. What is the complexity of MinSum k-DP over DAGs? Can this problem be solved as quickly as k-DSP over DAGs?

Chapter 9

Conclusion

> . . . these waters . . . heap themselves on me; they sweep me between their great shoulders; I am turned; I am tumbled; I am stretched, among these long lights, these long waves . . .
>
> Virginia Woolf, *Waves*

> . . . an ocean of knowledge is apt to drown you long before it educates you. The art of learning [is] in selection . . .
>
> Mark Lawrence, *The Book That Wouldn't Burn*

> I want to *recognize something I never saw before*. I want the vision to leap out at me, terrible and blazing—the fire of the transfiguring imagination.
>
> Urusula K. Le Guin, *The Wave in the Mind*

In this thesis, we discussed algorithms for parameterized relaxations of the circuit analysis problem, Majority-SAT, and the disjoint paths problems, All-Pairs Connectivity and 2-Disjoint Shortest Paths, as well as interesting variants of these tasks. For each of these starting problems, we observed that the tasks were intractable in a formal complexity-theoretic sense, and then argued that suitable relaxations of these problems could be solved faster, thereby helping us evade intractability.

Although Majority-SAT is unlikely to admit a polynomial-time algorithm (under the hypothesis that P $\not\subseteq$ PP, a weaker hypothesis than P $\not\subseteq$ NP) we showed that for any fixed integer $k \quad 1$, Majority-SAT can be solved in linear time over k-CNF formulas. Similarly, although All-Pairs Connectivity is unlikely to admit a truly subcubic time algorithm (under the Strong Exponential Time Hypothesis and 4-Clique Hypothesis), we showed that for any integer $k \quad 1$, the relaxation k-APC can be solved in $O((kn)^{!})$ time, which is optimal for all constant k (assuming a hardness hypothesis concerning the complexity of the Boolean Matrix Multiplication problem). Finally, we showed that 2-Disjoint Shortest Paths can be solved in linear time, despite the k-Disjoint Shortest Paths paths problem being NP-hard for general k.

The journey to these results has unveiled interesting properties of circuits and graphs. In our discussion of relaxations of Majority-SAT, we proved a *regularity lemma* for k-CNF formulas in Theorem 3.28. In our study of relaxations of APC, we showed how *low-rank enumeration* suffices to encode small collections of edge-disjoint paths. In our exploration of 2-Disjoint Shortest Paths, we saw how the enumerative properties of pairs of shortest paths in directed acyclic graphs and undirected graphs simplify modulo two.

Many beautiful mysteries remain—we refer the reader to the open problems raised in Chapter 5 as well as Sections 7.4 and 8.7 for potential avenues to investigate the material discussed in this thesis further. Beyond the specific open questions identified in those sections, the paradigm of parameterized relaxations is quite broad, and there are many other tasks for which this framework could prove useful.

To conclude, rather than discussing additional specific problems, we highlight one general research agenda, motivated by the questions in this work, which is: *to what extent can we combine combinatorial and algebraic techniques* for solving computational problems?

In Part I, we proved structural properties of k-CNFs and their satisfaction probabilities for constant k, to derive fast algorithms for kSAT-Prob$_p$. Although our proof techniques seem quite combinatorial, the arguments we employ brush up against areas where algebraic methods have proven useful. For example, our kSAT-Prob$_p$ algorithm in Section 3.3 utilizes an algorithm of [JZC04] as a subroutine, recorded in Proposition 3.15, for the $(k;s)$-Set Packing problem (a parameterized problem related to finding disjoint sets from a collection). There have been many subsequent algebraic algorithms for faster parameterized set packing, closely tied to computation on linear matroids [Kou05, EKW24]. Can these algebraic techniques help design faster exact parameterized algorithms for kSAT-Prob$_p$ and its variants?

In Part II we designed fast algorithms for relaxations of All-Pairs Connectivity and Disjoint Shortest Paths. Although our methods were algebraic, the fastest algorithms for many closely related problems, such as Maximum Flow and k-Disjoint Shortest Paths on directed acyclic graphs for $k \geq 3$, use combinatorial and optimization-based techniques. In some cases, as with the k-Vertex Connectivity problem, the fastest algorithms for these problems used to be algebraic, and were only recently superseded by combinatorial approaches.

In general, it seems like the key to obtaining faster algorithms for various problems related to circuits and graphs may require us to develop a better understanding of how to *combine* more traditional combinatorial techniques with algebraic frameworks.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, April 2009. doi:10.1017/cbo9780511804090.

[AGI+18] Amir Abboud, Loukas Georgiadis, Giuseppe F. Italiano, Robert Krauthgamer, Nikos Parotsidis, Ohad Trabelsi, Przemysław Uznański, and Daniel Wolleb-Graf. Faster Algorithms for All-Pairs Bounded Min-Cuts, 2018. arXiv:1807.05803v2.

[AGI+19] Amir Abboud, Loukas Georgiadis, Giuseppe F. Italiano, Robert Krauthgamer, Nikos Parotsidis, Ohad Trabelsi, Przemysław Uznański, and Daniel Wolleb-Graf. Faster Algorithms for All-Pairs Bounded Min-Cuts. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2019.7.

[AJ24] Shyan Akmal and Ce Jin. An Efficient Algorithm for All-Pairs Bounded Edge Connectivity. *Algorithmica*, January 2024. doi:10.1007/s00453-023-01203-2.

[Akh20] Maxim Akhmedov. Faster 2-Disjoint-Shortest-Paths Algorithm. In *Computer Science – Theory and Applications*, pages 103–116. Springer International Publishing, 2020. doi:10.1007/978-3-030-50026-9_7.

[AKL+22] Amir Abboud, Robert Krauthgamer, Jason Li, Debmalya Panigrahi, Thatchaphol Saranurak, and Ohad Trabelsi. Breaking the Cubic Barrier for All-Pairs Max-Flow: Gomory-Hu Tree in Nearly Quadratic Time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 884–895, 2022. doi:10.1109/FOCS54457.2022.00088.

[Akm24] Shyan Akmal. An Enumerative Perspective on Connectivity. In *2024 Symposium on Simplicity in Algorithms (SOSA)*, pages 179–198, 2024. doi:10.1137/1.9781611977936.18.

[AKT20] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. *New Algorithms and Lower Bounds for All-Pairs Max-Flow in Undirected Graphs*, page 48–61. Society for Industrial and Applied Mathematics, January 2020. doi:10.1137/1.9781611975994.4.

[APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, March 1979. doi:10.1016/0020-0190(79)90002-4.

[AW22] Shyan Akmal and Ryan Williams. MAJORITY-3SAT (and Related Problems) in Polynomial Time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1033–1043, 2022. doi:10.1109/FOCS52979.2021.00103.

[AWW24] Shyan Akmal, Virginia Vassilevska Williams, and Nicole Wein. Detecting Disjoint Shortest Paths in Linear Time and More, 2024. arXiv:2404.15916v2.

[AZ18] Martin Aigner and Günter M. Ziegler. *Proofs from THE BOOK*. Springer Berlin Heidelberg, 2018. doi:10.1007/978-3-662-57265-8.

[BDK01] Delbert D. Bailey, Víctor Dalmau, and Phokion G. Kolaitis. Phase Transitions of PP-Complete Satisfiability Problems. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 183–192. Morgan Kaufmann, 2001.

[BDK07] Delbert D. Bailey, Víctor Dalmau, and Phokion G. Kolaitis. Phase transitions of PP-complete satisfiability problems. *Discrete Applied Mathematics*, 155(12):1627–1639, June 2007. doi:10.1016/j.dam.2006.09.014.

[BDPR19] Florian Bridoux, Nicolas Durbec, Kevin Perrot, and Adrien Richard. Complexity of maximum fixed point problem in boolean networks. In *Computing with Foresight and Industry*, pages 132–143, Cham, 2019. Springer International Publishing.

[BDPR20] Florian Bridoux, Amélia Durbec, Kévin Perrot, and Adrien Richard. Complexity of fixed point counting problems in Boolean Networks. *CoRR*, abs/2012.02513, 2020. arXiv:2012.02513.

[Ben95] András A. Benczúr. Counterexamples for Directed and Node Capacitated Cut-Trees. *SIAM Journal on Computing*, 24(3):505–510, 1995. doi:10.1137/S0097539792236730.

[BFG24] Matthias Bentert, Fedor V. Fomin, and Petr A. Golovach. Tight Approximation and Kernelization Bounds for Vertex-Disjoint Shortest Paths, 2024. arXiv:2402.15348.

[BGK+23] Tatiana Belova, Alexander Golovnev, Alexander S. Kulikov, Ivan Mihajlin, and Denil Sharipov. *Polynomial formulations as a barrier for reduction-based hardness proofs*, page 3245–3281. Society for Industrial and Applied Mathematics, January 2023. doi:10.1137/1.9781611977554.ch124.

[BH74] James R. Bunch and John E. Hopcroft. Triangular Factorization and Inversion by Fast Matrix Multiplication. *Mathematics of Computation*, 28(125):231–236, 1974. doi:10.1090/s0025-5718-1974-0331751-8.

[BH19] Andreas Björklund and Thore Husfeldt. Shortest Two Disjoint Paths in Polynomial Time. *SIAM Journal on Computing*, 48(6):1698–1710, January 2019. doi:10.1137/18m1223034.

[BHK22] Andreas Björklund, Thore Husfeldt, and Petteri Kaski. The Shortest Even Cycle Problem is Tractable. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, June 2022. doi:10.1145/3519935.3520030.

[BHT12] Andreas Björklund, Thore Husfeldt, and Nina Taslaman. Shortest Cycle Through Specified Elements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, January 2012. doi:10.1137/1.9781611973099.139.

[BK17] Kristof Berczi and Yusuke Kobayashi. The Directed Disjoint Shortest Paths Problem. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ESA.2017.13.

[BNRZ21] Matthias Bentert, André Nichterlein, Malte Renken, and Philipp Zschoche. Using a Geometric Lens to Find k Disjoint Shortest Paths. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:14, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.26.

[BS83] Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22(3):317–330, February 1983. doi:10.1016/0304-3975(83)90110-x.

[BW24] Aaron Bernstein and Nicole Wein. Closing the Gap Between Directed Hopsets and Shortcut Sets, 2024. arXiv:2207.04507v4.

[CDdB16] İsmail İlkan Ceylan, Adnan Darwiche, and Guy Van den Broeck. Open-world probabilistic databases. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, pages 339–348. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12908.

[CFK+16] Marek Cygan, Fedor V Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer International Publishing, Cham, Switzerland, October 2016.

[CGI+16] Marco L. Carmosino, Jiawei Gao, Russell Impagliazzo, Ivan Mihajlin, Ramamohan Paturi, and Stefan Schneider. Nondeterministic Extensions of the Strong Exponential Time Hypothesis and Consequences for Non-reducibility. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, ITCS'16. ACM, January 2016. doi:10.1145/2840728.2840746.

[Chi23] Rajesh Chitnis. A tight lower bound for edge-disjoint paths on planar dags. *SIAM Journal on Discrete Mathematics*, 37(2):556–572, May 2023. URL: http://dx.doi.org/10.1137/21M1395089, doi:10.1137/21m1395089.

[CKL13] Ho Yee Cheung, Tsz Chiu Kwok, and Lap Chi Lau. Fast Matrix Rank Algorithms and Applications. *J. ACM*, 60(5), oct 2013. doi:10.1145/2528404.

[CKL+22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623, 2022. doi:10.1109/FOCS54457.2022.00064.

[CLL13] Ho Yee Cheung, Lap Chi Lau, and Kai Man Leung. Graph Connectivities, Network Coding, and Expander Graphs. *SIAM Journal on Computing*, 42(3):733–751, 2013. doi:10.1137/110844970.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[CM18] Fabio G. Cozman and Denis D. Mauá. The complexity of Bayesian networks specified by propositional and relational languages. *Artificial Intelligence*, 262:96–141, 2018. doi:10.1016/j.artint.2018.06.001.

[CQ21] Chandra Chekuri and Kent Quanrud. Faster Algorithms for Rooted Connectivity in Directed Graphs. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49:1–49:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.49.

[CR94] Joseph Cheriyan and John H. Reif. Directed *s*-*t* Numberings, Rubber Bands, and Testing Digraph *k*-Vertex Connectivity. *Combinatorica*, 14(4):435–451, December 1994. doi:10.1007/bf01302965.

[Dar09] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009. URL: http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521884389.

[Dar21] Adnan Darwiche. Beyond NP with Tractable Circuits. Beyond Satisfiability Workshop at the Simon's Institute, 2021. URL: https://simons.berkeley.edu/talks/beyond-np-tractable-circuits.

[Din16] Irit Dinur. Mildly exponential reduction from gap-3sat to polynomial-gap label-cover. *Electronic colloquium on computational complexity ECCC ; research reports, surveys and books in computational complexity*, August 2016.

[DW22] Heiko Dietrich and James B. Wilson. Group isomorphism is nearly-linear time for most orders. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 457–467, 2022. doi:10.1109/FOCS52979.2021.00053.

[EG04] Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, 326(1–3):57–67, October 2004. doi:10.1016/j.tcs.2004.05.009.

[EKW24] Eduard Eiben, Tomohiro Koana, and Magnus Wahlström. *Determinantal Sieving*, page 377–423. Society for Industrial and Applied Mathematics, January 2024. doi:10.1137/1.9781611977912.16.

[ER60] P. Erdös and R. Rado. Intersection Theorems for Systems of Sets. *Journal of the London Mathematical Society*, s1-35(1):85–90, January 1960. doi:10.1112/jlms/s1-35.1.85.

[ET98] Tali Eilam-Tzoreff. The disjoint shortest paths problem. *Discrete Applied Mathematics*, 85(2):113–138, June 1998. doi:10.1016/s0166-218x(97)00121-2.

[FGL12] Pierluigi Frisco, Gordon Govan, and Alberto Leporati. Asynchronous P systems with active membranes. *Theoretical Computer Science*, 429:74–86, 2012. Magic in Science. doi:10.1016/j.tcs.2011.12.026.

[FHW80] Steven Fortune, John Hopcroft, and James Wyllie. The Directed Subgraph Homeomorphism Problem. *Theoretical Computer Science*, 10(2):111–121, February 1980. doi:10.1016/0304-3975(80)90009-2.

[FLSZ18] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, December 2018. doi:10.1017/9781107415157.

[FM71] M. J. Fischer and A. R. Meyer. Boolean Matrix Multiplication and Transitive Closure. In *12th Annual Symposium on Switching and Automata Theory (SWAT 1971)*. IEEE, October 1971. doi:10.1109/swat.1971.4.

[FNY+20] Sebastian Forster, Danupon Nanongkai, Liu Yang, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. *Computing and Testing Small Connectivity in Near-Linear Time and Queries via Fast Local Cut Algorithms*, page 2046–2065. Society for Industrial and Applied Mathematics, January 2020. doi:10.1137/1.9781611975994.126.

[Fra11] Andras Frank. *Connections in Combinatorial Optimization*. Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press, London, England, February 2011.

[GGI+17] Loukas Georgiadis, Daniel Graf, Giuseppe F. Italiano, Nikos Parotsidis, and Przemysław Uznański. All-Pairs 2-Reachability in $O(n^! \log n)$ Time. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 74:1–74:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2017.74.

[GHM08] Judy Goldsmith, Matthias Hagen, and Martin Mundhenk. Complexity of DNF minimization and isomorphism testing for monotone formulas. *Information and Computation*, 206(6):760–775, 2008. doi:10.1016/j.ic.2008.03.002.

[Gil74] John T. Gill. Computational Complexity of Probabilistic Turing Machines. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, page 91–95, New York, NY, USA, 1974. Association for Computing Machinery. doi:10.1145/800119.803889.

[GJ79] Michael R Garey and David S Johnson. *Computers and intractability*. W.H. Freeman, New York, NY, April 1979.

[GLR+23] Venkatesan Guruswami, Bingkai Lin, Xuandi Ren, Yican Sun, and Kewen Wu. Parameterized Inapproximability Hypothesis under ETH, 2023. arXiv:2311.16587.

[Gow97] W.T. Gowers. Lower bounds of tower type for Szemerédis uniformity lemma. *Geometric and Functional Analysis*, 7(2):322–337, May 1997. doi:10.1007/pl00001621.

[HL23] Xiaoyu He and Ray Li. Approximating Binary Longest Common Subsequence in Almost-Linear Time. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC '23. ACM, June 2023. doi:10.1145/3564246.3585104.

[HLSW23] Zhiyi Huang, Yaowei Long, Thatchaphol Saranurak, and Benyu Wang. Tight Conditional Lower Bounds for Vertex Connectivity Problems. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC '23. ACM, June 2023. doi:10.1145/3564246.3585223.

[IMH82] Oscar H Ibarra, Shlomo Moran, and Roger Hui. A Generalization of the Fast LUP Matrix Decomposition Algorithm and Applications. *Journal of Algorithms*, 3(1):45–56, March 1982. doi:10.1016/0196-6774(82)90007-4.

[JMV18] Hamidreza Jahanjou, Eric Miles, and Emanuele Viola. Local reduction. *Information and Computation*, 261:281–295, August 2018. doi:10.1016/j.ic.2018.02.009.

[JX24] Ce Jin and Yinzhan Xu. Shaving Logs via Large Sieve Inequality: Faster Algorithms for Sparse Convolution and More, 2024. arXiv:2403.20326.

[JZC04] Weijia Jia, C huanlin Zhang, and Jianer Chen. An efficient parameterized algorithm for m-set packing. *Journal of Algorithms*, 50(1):106–117, January 2004. doi:10.1016/j.jalgor.2003.07.001.

[KdC15a] Johan Kwisthout and Cassio P. de Campos. Computional complexity of Bayesian networks, July 2015. Tutorials of the 31st Conference on Uncertainty in Artificial Intelligence. URL: https://www.youtube.com/watch?v=7CU5uo2XwIc.

[KdC15b] Johan Kwisthout and Cassio P. de Campos. Lecture notes: Computational complexity of Bayesian networks, July 2015. Tutorials of the 31st Conference on Uncertainty in Artificial Intelligence. URL: https://auai.org/uai2015/proceedings/slides/UAI2015_Comp_LN.pdf.

[KG05] Andreas Krause and Carlos Guestrin. Optimal nonmyopic value of information in graphical models - efficient algorithms and theoretical limits. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1339–1345. Professional Book Center, 2005. URL: http://ijcai.org/Proceedings/05/Papers/1154.pdf.

[KG09] Andreas Krause and Carlos Guestrin. Optimal value of information in graphical models. *Journal of Artificial Intelligence Research*, 35:557–591, July 2009. doi:10.1613/jair.2737.

[KKR12] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424–435, March 2012. doi:10.1016/j.jctb.2011.07.004.

[Kou05] Ioannis Koutis. A faster parameterized algorithm for set packing. *Information Processing Letters*, 94(1):7–9, April 2005. doi:10.1016/j.ipl.2004.12.005.

[KPS24] Tuukka Korhonen, Michał Pilipczuk, and Giannos Stamoulis. Minor Containment and Disjoint Paths in almost-linear time, 2024. arXiv:2404.03958.

[KT18] Robert Krauthgamer and Ohad Trabelsi. Conditional Lower Bounds for All-Pairs Max-Flow. *ACM Trans. Algorithms*, 14(4), aug 2018. doi:10.1145/3212510.

[Kün18] Marvin Künnemann. On Nondeterministic Derandomization of Freivalds' Algorithm: Consequences, Avenues and Algorithmic Progress. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 56:1–56:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2018.56.

[Kwi11] Johan Kwisthout. Most probable explanations in Bayesian networks: Complexity and tractability. *International Journal of Approximate Reasoning*, 52(9):1452–1469, 2011. Handling Incomplete and Fuzzy Information in Data Analysis and Decision Processes. doi:10.1016/j.ijar.2011.08.003.

[LGM98] Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. The Computational Complexity of Probabilistic Planning. *Journal of Artificial Intelligence Research*, 9:1–36, August 1998. doi:10.1613/jair.505.

[Lin20] Andrea Lincoln. *Applications of Fine-Grained Complexity*. PhD thesis, Massachusetts Institute of Technology, 2020.

[LLW88] N. Linial, L. Lovász, and A. Wigderson. Rubber Bands, Convex Embeddings and Graph Connectivity. *Combinatorica*, 8(1):91–102, March 1988. doi:10.1007/bf02122557.

[LNP+21] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex Connectivity in Poly-logarithmic Max-Flows. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, page 317–329, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3406325.3451088.

[Loc21] Willian Lochet. A Polynomial Time Algorithm for the k-Disjoint Shortest Paths Problem. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 169–178. Society for Industrial and Applied Mathematics, January 2021. doi:10.1137/1.9781611976465.12.

[MDCC15] Denis D. Mauá, Cassio P. De Campos, and Fabio G. Cozman. The complexity of MAP inference in Bayesian networks specified through logical languages. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 889–895. AAAI Press, 2015.

[MP93] Matthias Middendorf and Frank Pfeiffer. On the complexity of the disjoint paths problem. *Combinatorica*, 13(1):97–107, March 1993. doi:10.1007/bf01202792.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, August 1995. doi:10.1017/cbo9780511814075.

[MS14] Guy Moshkovitz and Asaf Shapira. A short proof of Gowers' lower bound for the regularity lemma. *Combinatorica*, 36(2):187–194, November 2014. doi:10.1007/s00493-014-3166-4.

[Mun00a] Martin Mundhenk. The Complexity of Optimal Small Policies. *Mathematics of Operations Research*, 25(1):118–129, 2000. doi:10.1287/moor.25.1.118.15214.

[Mun00b] Martin Mundhenk. The Complexity of Planning with Partially-Observable Markov Decision Processes. Technical report, Dartmouth College, USA, 2000.

[MV97] Meena Mahajan and V. Vinay. Determinant: Combinatorics, algorithms, and complexity. *Chicago Journal of Theoretical Computer Science*, 1997.

[MW21] Abhijit S. Mudigonda and R. Ryan Williams. Time-Space Lower Bounds for Simulating Proof Systems with Quantum and Randomized Verifiers. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference (ITCS 2021)*, volume 185 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 50:1–50:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITCS.2021.50.

[Pap93] Christos H Papadimitriou. *Computational Complexity*. Pearson, Upper Saddle River, NJ, November 1993.

[PD04] James D. Park and Adnan Darwiche. Complexity Results and Approximation Strategies for MAP Explanations. *J. Artif. Intell. Res.*, 21:101–133, 2004. doi:10.1613/jair.1236.

[PLMZ11] Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. P systems with elementary active membranes: Beyond np and conp. In *Membrane Computing*, pages 338–347, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Rao22] Anup Rao. Sunflowers: From Soil to Oil. *Bulletin of the American Mathematical Society*, 60(1):29–38, September 2022. doi:10.1090/bull/1777.

[RS95] N. Robertson and P.D. Seymour. Graph Minors .XIII. The Disjoint Paths Problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, January 1995. doi:10.1006/jctb.1995.1006.

[Sch02] Alexander Schrijver. *Combinatorial Optimization*. Algorithms and Combinatorics. Springer, Berlin, Germany, 2003 edition, December 2002.

[SCWW21] Jessica Su, Kathy Cooper, Nicole Wein, and Virginia Vassilevska Williams. MIT 6.890 Lecture Notes: Lecture 1. https://people.csail.mit.edu/virgi/6.890/lecture1.pdf, September 2021.

[Sim75] Janos Simon. *On Some Central Problems in Computational Complexity*. PhD thesis, Cornell University, January 1975. URL: https://ecommons.cornell.edu/handle/1813/6975.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. Wadsworth Publishing, Belmont, CA, 3 edition, June 2012.

[SY09] Amir Shpilka and Amir Yehudayoff. Arithmetic Circuits: A survey of recent results and open questions. *Foundations and Trends® in Theoretical Computer Science*, 5(3-4):207–388, 2009. doi:10.1561/0400000039.

[Sze75] E. Szemerédi. On sets of integers containing k elements in arithmetic progression. *Acta Arithmetica*, 27(1):199–245, 1975. URL: http://eudml.org/doc/205339.

[Tan22a] Till Tantau. On the Satisfaction Probability of k-CNF Formulas. In Shachar Lovett, editor, *37th Computational Complexity Conference (CCC 2022)*, volume 234 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:27, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CCC.2022.2.

[Tan22b] Till Tantau. On the Satisfaction Probability of k-CNF Formulas, 2022. arXiv:2201.08895v3.

[TF10] Tino Teige and Martin Fränzle. Resolution for stochastic boolean satisfiability. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 625–639, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Tho97] M. Thorup. Undirected Single Source Shortest Paths in Linear Time. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc, 1997. doi:10.1109/sfcs.1997.646088.

[Tho05] Torsten Tholey. Solving the 2-Disjoint Paths Problem in Nearly Linear Time. *Theory of Computing Systems*, 39(1):51–78, November 2005. doi:10.1007/s00224-005-1256-9.

[Tho12] Torsten Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theoretical Computer Science*, 465:35–48, December 2012. doi:10.1016/j.tcs.2012.09.025.

[Tod91] Seinosuke Toda. PP is as Hard as the Polynomial-Time Hierarchy. *SIAM Journal on Computing*, 20(5):865–877, October 1991. doi:10.1137/0220053.

[Tra23] Ohad Trabelsi. (Almost) Ruling Out SETH Lower Bounds for All-Pairs Max-Flow, 2023. arXiv:2304.04667v3.

[Val79a] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. doi:10.1137/0208032.

[Val79b] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979. doi:10.1016/0304-3975(79)90044-6.

[Vis13] Nisheeth K Vishnoi. $Lx = b$. Foundations and Trends (R) in Theoretical Computer Science. now, Hanover, MD, 3 edition, May 2013.

[VW21] Nikhil Vyas and Ryan Williams. On Super Strong ETH. *Journal of Artificial Intelligence Research*, 70:473–495, January 2021. doi:10.1613/jair.1.11859.

[Wag86] Klaus W. Wagner. The Complexity of Combinatorial Problems with Succinct Input Representation. *Acta Informatica*, 23(3):325–356, June 1986. doi:10.1007/bf00289117.

[Wei21] Nicole Spence Wein. *Algorithms and Hardness for Approximating the Diameter of a Graph*. PhD thesis, Massachusetts Institute of Technology, September 2021.

[Wig19] Avi Wigderson. *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press, October 2019. doi:10.2307/j.ctvckq7xb.

[Wil13] Ryan Williams. Improving Exhaustive Search Implies Superpolynomial Lower Bounds. *SIAM Journal on Computing*, 42(3):1218–1244, January 2013. doi:10.1137/10080703x.

[Wil24] Ryan Williams. Self-Improvement for Circuit-Analysis Problems. *Electronic Colloquium Computational Complexity*, TR23-082 Revision #1, 2024. URL: https://eccc.weizmann.ac.il/report/2023/082.

[WXXZ24] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. *New Bounds for Matrix Multiplication: from Alpha to Omega*, page 3792–3835. Society for Industrial and Applied Mathematics, January 2024. doi:10.1137/1.9781611977912.134.

[Zuc96] David Zuckerman. On Unapproximable Versions of NP-Complete Problems. *SIAM Journal on Computing*, 25(6):1293–1304, 1996. doi:10.1137/S0097539794266407.