THEORETICAL FOUNDATIONS FOR PRACTICAL PROBLEMS: NETWORK ANALYSIS, ALGORITHM SELECTION, AND INTERACTION DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Rishi Vijay Gupta
December 2016

Abstract

Over the past forty years, worst-case analysis has emerged as the dominant framework for analyzing problems in theoretical computer science. While successful on many fronts, there are domains where worst-case analysis is not guiding algorithmic development in a direction useful for solving problems in practice. The proximate cause is that the parts of the input space directly exposed to worst-case analyses (the "worst-case instances" of the best algorithms) are not representative of instances typically seen in practice.

In this work, we look at three domains where it is easy to see that a worst-case framework is not appropriate, but where the structure of instances seen in practice is poorly understood, and hence hard to take advantage of directly. For each domain, we propose a framework that captures some part of this structure, along with new algorithms and theorems inspired by our framework. The three domains are:

• Network analysis. We consider the class of triangle-dense graphs, which include most social and biological networks. We show that triangle-dense graphs resemble unions of cliques in a quantifiable sense, and exhibit new clustering and clique-counting algorithms that take advantage of that structure.

• Application specific algorithm selection. Given a computational problem, and a set of already implemented algorithms, which one should you run? We present one framework that models algorithm selection as a statistical learning problem, and show that one can transfer dimension notions from statistical learning theory to algorithm selection. We also study the online version of the algorithm selection problem, and give possibility and impossibility results for the existence of no-regret learning algorithms.

• Interaction design. We look at a specific problem in understanding how users are interacting with an application, given traces of their behavior. We model the problem as that of recovering a mixture of Markov chains, and show that under mild assumptions the mixture can be recovered from only its induced distribution on consecutive triples of states.

Acknowledgments

First and foremost, I would like to thank my advisor, Tim Roughgarden, for being everything that an advisor can be. I am continually inspired by Tim's patience, clarity of thought, the deliberateness with which he chooses what he works on, and his incredible level of investment in undergrads, grad students, and junior faculty. It was also a privilege to be a part of Tim's day to day life, and experience first hand Tim's speed, stamina, and relentless pursuit of truth.

I had the fortune of having several other advisors from first-year rotations and summer internships. I would particularly like to thank Deeparnab Chakrabarty, Ashish Goel, Ramesh Johari, and Sergei Vassilvitskii for shaping how I think about research and how I think about being a researcher. I would also like to thank Satish Rao for humoring my extended stay at Berkeley. I would like to thank Tim's academic family, and Kshipra Bhawalkar and Mukund Sundararajan in particular, for always having their door open to me, and for having a keen understanding of graduate school and life.

My research would not have been possible without the guidance and sweat of many different collaborators; I would like to thank Eric Price, Yaron Rachlin, C. Seshadhri, Josh Wang, and Virginia Williams for particularly productive collaborations. I would also like to thank the CS administrators for expertly knocking down every barrier they could between the graduate students and their research.

Graduate school is one step in the middle of a journey, and there are many people who made it possible for me to arrive at graduate school at all. There are too many individuals to name, but in particular I would like to thank my M.Eng. advisor Piotr Indyk and Maria Monks for giving me the skills and credentials needed to apply for graduate school, and for encouraging me to make it happen. I would also like to thank the institutions and the individuals behind them that quietly make such journeys possible. The Gunn Robotics Team, the Math Olympiad Summer Program, and East Campus and Random Hall from the MIT dormitory system all had an outsized influence on my path as a scientist.

Last but not least, graduate school would not have been possible without the intellectual and emotional support of family and friends. At minimum I would like to thank the many amazing students in the Berkeley and Stanford theory groups, and the students I lived with at Little Mountain and Ayrshire Farm. Finally, I'd like to thank my parents for instilling an early love of mathematics, and my parents and sister for their enduring love and support.

Contents

Abstract iv

Acknowledgments v

1 Introduction 1

1.1 Interaction Networks 2

1.2 Application-Specific Algorithm Selection 4

1.3 Session Types in Interaction Design 6

1.4 A Note on Published Works 7

2 A New Framework for Interaction Networks 8

2.1 Introduction 8

Triangle-Dense Graphs 9

Main Results 10

Model Discussion 12

2.2 An Intuitive Overview 14

2.3 The Decomposition Procedure 15

Cleaning a Graph 15

Extracting a Single Cluster 16

Preserving Edges 20

2.4 Triangle-Dense Graphs: The Rogues' Gallery 23

2.5 Recovering Everything When ϵ is High 25

2.6 Recovering a Planted Clustering 26

2.7 Experiments in Counting Small Cliques 29

Previous Work in Clique Counting 30

The CES Algorithm 31

Empirical Evaluation 32

2.8 Conclusions and Further Directions 36

2.A Implementation Details 39

3 Application-Specific Algorithm Selection **43**
3.1 Introduction . . . 43
3.2 Motivating Scenarios . . . 45
Greedy Heuristic Selection . . . 45
Self-Improving Algorithms . . . 45
Parameter Tuning in Optimization and Machine Learning . . . 46
Empirical Performance Models for SAT Algorithms . . . 46
3.3 PAC Learning an Application-Specific Algorithm . . . 47
The Basic Model . . . 47
Pseudo-Dimension and Uniform Convergence . . . 48
Application: Greedy Heuristics and Extensions . . . 49
Application: Self-Improving Algorithms Revisited . . . 55
Application: Feature-Based Algorithm Selection . . . 57
Application: Choosing the Step Size in Gradient Descent . . . 59
3.4 Online Learning of Application-Specific Algorithms . . . 64
The Online Learning Model . . . 64
An Impossibility Result for Worst-Case Instances . . . 65
A Smoothed Analysis . . . 67
3.5 Conclusions and Future Directions . . . 70

4 Learning Session Types from Traces **71**
4.1 Introduction . . . 71
4.2 Preliminaries . . . 73
4.3 Conditions for Unique Reconstruction . . . 74
4.4 A Reconstruction Algorithm . . . 75
4.5 Analysis . . . 78

Bibliography **80**

Chapter 1

Introduction

Characterizing algorithm performance is a tricky business. For most problems, any given algorithm is going to have widely varying performance over the possible inputs to the problem, and the performance of any two reasonable algorithms is going to be incomparable over the input space. Rather than trying to directly reason about the performance of an algorithm on the input space as a whole, a typical analysis will summarize it as a function of just one or two parameters (e.g. the input size). An example of such a summary might be a worst-case running time, or an average-case running time with respect to a distribution.

Though seemingly innocuous, an analysis framework, such as a summarizing statistic, is of great importance in guiding algorithmic development for a particular problem. Algorithm designers naturally optimize to accepted metrics, and different metrics emphasize different parts of the input space. Examples of frameworks that have spurred new developments include smoothed analysis, competitive analysis, and various semi-random models; see the lecture notes at [Rou14] for an introduction to many more.

Over the past forty years, worst-case analysis has emerged as the dominant framework for analyzing problems in theoretical computer science. While leading to a beautiful theory of complexity, and enjoying many algorithmic successes, there are many domains where worst-case analysis isn't able to guide development in a direction useful for solving problems in practice. The proximate cause is usually that the parts of the input space directly exposed to worst-case analyses (the "worst-case instances" of the best algorithms) are somehow not representative of the types of instances seen in practice.

In this dissertation, we look at three problem domains where it is easy to see that a worst-case framework is not appropriate, but where the structure of instances seen in practice is poorly understood, and hence hard to exploit directly. For each domain, we propose a framework that captures some part of this structure, along with new algorithms and theorems inspired by the framework.

The rest of the dissertation is organized as follows. We briefly introduce the three problem domains in the following three sections, and provide a sample result from each. We end the chapter with a note on published works. Chapters 2, 3 and 4 tackle each problem in detail. Each of the 4 chapters is completely self-contained.

1.1 Interaction Networks

There are many problems on graphs that are NP-hard, but which are often tractable in practice. For instance, any reasonable formalization of clustering results in an NP-hard problem (e.g. by including planted clique as a special case), but people regularly cluster graphs with hundreds of millions of nodes. This motivates the search for a framework for analyzing graph algorithms that emphasizes a more practical set of inputs.

In Chapter 2, we focus on finding such a framework for the class of *interaction networks*, or interaction graphs. Examples of such graphs include

- Social networks, such as the Facebook graph (nodes are people, and edges are friendships), or the Wikipedia voting graph (nodes are elected administrators, and two nodes share an edge if one voted for the other).
- Biological networks, such as protein-protein interaction networks (nodes are proteins, and two proteins share an edge if they physically interact in a biological pathway).
- Systemic networks, such as interbank payment networks (nodes are banks, and two banks share an edge if one bank has directly transferred money to the other in the last 12 months).

What makes interaction networks special is that if node A is connected to node B, and node B is connected to node C, then node A is connected to node C with much higher probability than would be suggested by the edge density of the network. Non-examples of interaction networks include road networks (streets are nodes, and two streets share an edge if they intersect), or random Erdős-Rényi graphs.

Given the general interest in interaction networks, a lot of work has already been done to try to understand what makes them special. Much of the recent work on such networks has focused on developing generative models (along with average-case analyses) that produce graphs with statistical properties similar to those found in practice.

We pursue a different approach. In lieu of adopting a particular generative model, we ask:

> *Is there a combinatorial assumption weak enough to hold in every "reasonable" model of interaction networks, yet strong enough to permit useful structural and algorithmic results?*

In other words, we seek a framework that will apply to *every* reasonable model of interaction networks, including those yet to be devised, that is yet limited enough to not include the worst-case examples. We explore the problem of clustering a graph to initiate the discussion, but we believe our proposal is useful for other graph computations as well.

Triangle-Dense Graphs

Let a *wedge* be a two-hop path in an undirected graph. The vertices of a wedge form an *open* wedge if they induce a path, and a *closed* wedge if they induce a triangle. We now define the key parameter we will use to differentiate the graphs of interest from the worst-case graphs.

Definition 1.1 (Triangle Density). *The* triangle density *of an undirected graph G is the fraction of wedges in G that are closed.*

Alternatively, it is 3 times the number of triangles in G divided by the number of wedges in G. Since every triangle of a graph contains three wedges, and no two triangles share a wedge, the triangle density of a graph is between 0 and 1.

As an example, the triangle density of a graph is 1 if and only if it is the union of cliques. The triangle density of a random Erdős-Rényi graph is concentrated around its edge density.

Interaction networks are generally sparse, and yet have remarkably high triangle density. The Facebook graph, for instance, has triangle density 0.16 [UKBM11], whereas an Erdős-Rényi graph with the same edge density would have triangle density less than 10^{-6}. Large triangle density, meaning triangle density much higher that what the edge density would suggest, is perhaps one of the most widely cited signatures of social and interaction networks (see related work in Chapter 2).

In the next section, we show that having large triangle density already implies a surprising amount of structure. Though social scientists and others have been measuring triangle density in networks for many years, the idea of parameterizing algorithm analysis by triangle density is new to this work.

Sample Result: A Decomposition Theorem

As noted above, the triangle density of a graph is 1 if and only if it is a union of cliques. More surprisingly, we show that large triangle density is enough to ensure that a graph is approximately a union of cliques.

We first formally define what we mean by an approximate union of cliques.

Definition 1.2 (Tightly-Knit Family). *Let $\rho > 0$. A collection $G_1, G_2, \ldots, G_k$ of vertex disjoint subgraphs of a graph G forms a ρ-*tightly-knit family *if*

- *Each G_i has both edge density and triangle density at least ρ.*
- *Each G_i has radius at most 2.*

There is no a priori restriction on the number of subgraphs k.

Our main theorem states that every graph with constant triangle density contains a tightly-knit family that captures a constant fraction of the graph's triangles (with the constants depending on the triangle density).

Theorem 1.3 (Decomposition of Triangle-Dense Graphs). *There exists a polynomial $f(\epsilon) = \epsilon^c$ for some constant $c > 0$, such that for every graph G with triangle density at least ϵ, there exists an $f(\epsilon)$-tightly-knit family that contains an $f(\epsilon)$ fraction of the triangles of G.*

Graphs without constant triangle density, such as sparse Erdős-Rényi random graphs, do not generally admit non-trivial tightly-knit families, even if the triangle density requirement for each cluster is dropped.

In addition to the structural result, we present an efficient algorithm for finding such a decomposition, which is of independent interest. A preliminary implementation of the algorithm requires a few minutes on a laptop to decompose networks with millions of edges, and a few simple heuristics would speed it up even further.

1.2 Application-Specific Algorithm Selection

Given a computational problem, and a number of different *already implemented* algorithms (say downloaded from the internet), how should one reason about picking an algorithm to run? Any worst-case formulation of this problem is likely to be non-computable, let alone NP-hard. Despite there being a large literature on empirical approaches to algorithm selection, there has been surprisingly little theoretical consideration of the problem.

In Chapter 3, we show that *application-specific algorithm selection* can be usefully modeled as a learning problem. We propose two different models, and show results in each. In the first model, inspired by PAC learning, the "application-specific" information is encoded as an unknown distribution $\mathcal{D}$ over inputs to the algorithm. After seeing some number of instances drawn from $\mathcal{D}$, the algorithm selector's goal is simply to pick an algorithm that does almost as well as the best algorithm in their set, (in expectation) with respect to $\mathcal{D}$. The second model is inspired by online learning, and is explained below.

The Online Learning Model

This section formalizes the problem of learning a best algorithm *online*, with instances arriving one-by-one. The goal is choose an algorithm at each time step, before seeing the next instance, so that the average performance is close to that of the best fixed algorithm in hindsight.

Formally, we have a computational or optimization problem Π (e.g., Maximum Weight Independent Set), a set $\mathcal{A}$ of algorithms for Π (e.g., a parameterized family of heuristics), and a performance measure $\textsc{cost} : \mathcal{A} \times \Pi \to [0, 1]$ (e.g., the total weight of the returned solution). We also have an unknown instance sequence $x_1, \ldots, x_T$.

A learning algorithm outputs a sequence $A_1, \ldots, A_T$ of algorithms. Each algorithm A_i is chosen (perhaps probabilistically) with knowledge only of the previous instances $x_1, \ldots, x_{i-1}$. The standard goal in online learning is to choose $A_1, \ldots, A_T$ to minimize the worst-case (over $x_1, \ldots, x_T$) *regret*, defined as the average performance loss relative to the best algorithm $A \in \mathcal{A}$ in hindsight (without loss of generality, assume $\textsc{cost}$ corresponds to a maximization objective):

$$\frac{1}{T}\left(\sup_{A\in\mathcal{A}}\sum_{t=1}^{T}\textsc{cost}(A,x_i)-\sum_{t=1}^{T}\textsc{cost}(A_i,x_i)\right). \tag{1.1}$$

A *no-regret* learning algorithm has expected (over its coin tosses) regret $o(1)$, as $T \to \infty$, for every instance sequence. The design and analysis of no-regret online learning algorithms is a mature field; for example, many no-regret online learning algorithms are known for the case of a finite set $|\mathcal{A}|$ (e.g. the "multiplicative weights" algorithms).

Sample Result: Online Learning of Independent Set

We first define the Maximum Weight Independent Set problem (MWIS), along with a parameterized family of heuristics. The input to MWIS is an undirected graph $G = (V, E)$ with a fixed number of vertices n, and a non-negative weight w_v for each vertex $v \in V$. The goal is to compute the independent set — a subset of mutually non-adjacent vertices — with maximum total weight. Two natural greedy heuristics are to greedily choose vertices (subject to feasibility) in order of non-increasing weight w_v, or nonincreasing density $w_v/(1 + \deg(v))$. The intuition for the denominator is that choosing v "uses up" $1 + \deg(v)$ vertices — v and all of its neighbors.

We define the family of algorithms $\mathcal{A} = \{A_\rho\}_{\rho\in[0,1]}$ as follows. The algorithm A_ρ chooses vertices, subject to feasibility, in order of nonincreasing $w_v/(1+\deg(v))^\rho$. Note that $\mathcal{A}$ smoothly interpolates between the two heuristics in the paragraph above. We first have the following negative result.

Theorem 1.4 (Impossibility of Online Learning MWIS)**.** *There is a distribution on MWIS input sequences over which every algorithm has expected regret* $1 - 1/\operatorname{poly}(n)$.

Recall that $1-1/\operatorname{poly}(n)$ is essentially the worst regret possible. Despite the negative result above, we show a "low-regret" learning algorithm for MWIS under a slight restriction on how the instances x_t are chosen. By low-regret we mean regret that is polynomially small in the number of vertices n.

We take the approach suggested by smoothed analysis [ST09]. Fix a parameter $0 < \sigma < 1$. We allow each x_t to have an arbitrary graph on n vertices, but only allow each weight w_v to be specified to its first $\log(1/\sigma)$ bits. The remaining lower order bits are set at random. This is a reasonable restriction on the algorithm's adversary if we imagine the inputs to be somewhat noisy in practice. We call such an instance a σ-smooth MWIS instance.[1] We then have the following result:

Theorem 1.5 (Online Learning of Smooth MWIS)**.** *There is a* $\operatorname{poly}(n, 1/\sigma)$ *time algorithm with expected regret* $1/\operatorname{poly}(n)$ *for* σ*-smooth MWIS.*

[1] The definition of σ-smooth MWIS in Chapter 3 is somewhat less restrictive than this, but this definition captures the significant bits.

1.3 Session Types in Interaction Design

The last scenario we look at is motivated by a very specific problem in interaction design. Given an app or website, we would like to understand how users are traveling through the app. The output of such an analysis might be a list of common flows, or a list of common but unexpected behaviors, due to e.g. users working around a user interface bug. We are not aware of any previous formulation of this task, either as an empirical or theoretical problem.

As a motivating example, consider the usage of a standard maps app on a phone. There are a number of different reasons one might come to the app: to search for a nearby business, to get directions from one point to another, or just to orient oneself. Once the app has some users, we would like to algorithmically discover things like:

- Common uses for the app that the designers hadn't expected, or hadn't expected to be common. For instance, maybe a good fraction of users (or user sessions) simply use the app to check the traffic.
- Whether different types of users use the app differently. For instance, maybe experienced users use the app differently than first time users, either due to having different goals, or due to accomplishing the same tasks more efficiently.
- Undiscoverable flows. For instance, maybe there is an "I'm hungry" tab in a drop down menu, but many users are instead using some convoluted path through the "Find a business" tab to search for food.
- Other unexpected behavior, such as users consistently pushing a button several times before the app responds, or doing other things to work around programming or UI errors.

We model the problem as follows. There are a small number L of unknown session types, such as "experienced user looking for food", or "first time user trying to figure out what the app does", or "user hitting bug #1234". There are $n \geq 2L$ observable states; the states can correspond to e.g. pages on a site, or actions (swipe, type, zoom) on an app. As an initial attempt to formalize the problem, we assume user behavior is Markovian for each fixed session type; namely, each of the L session types corresponds to a Markov chain on the n states.

The app designers receive session trails of the following form: a session type and starting state is selected from some unknown distribution over $L \times n$, and then the session follows the type-specific Markov chain for some number of steps. The computational problem is to recover the transition probabilities in each of the L individual Markov chains, given a list of these trails. The L Markov chains can then be passed to a human for examination, past and future traces can be labelled with their most likely type, or users can be bucketed based on their typical session types.

Mathematically, this problem corresponds to learning a mixture of Markov chains, and is of independent interest. The most closely related theoretical work is on learning Hidden Markov Models, learning phylogenetic trees, or learning mixtures of Hidden Markov Models.

Sample Result: Learning a Mixture of Markov Chains

Let $\mathcal{S}$ be the distribution over session type and starting state, and let $\mathcal{M}$ be the transition probabilities for the L Markov chains. Let a t-trail be a trail of length t, namely, a starting state drawn from $\mathcal{S}$ followed by $t-1$ steps in the corresponding Markov chain in $\mathcal{M}$. Our main result is that only 3-trails are needed for recovery of $\mathcal{M}$ and $\mathcal{S}$. Specifically,

Theorem 1.6 (Recovering a Mixture of Markov Chains from 3-Trails)**.** *Under appropriate non-degeneracy conditions, the mixture parameters $\mathcal{M}$ and $\mathcal{S}$ can be recovered exactly given their induced distribution over 3-trails.*

The non-degeneracy conditions are somewhat complicated, but benign. We also provide a fast algorithm for doing the recovery, and show that on synthetic data it does better than the natural expectation-maximization (EM) algorithm in some regimes.

1.4 A Note on Published Works

The results in Chapter 2 are joint work with Tim Roughgarden and C. Seshadhri, and many of them appear in [GRS16]. Section 2.7 is additionally joint work with Joshua Wang. Appendix 2.A appears in [GRS14]. Section 2.5 and Section 2.7 are new to this thesis, as is some of Section 2.8.

The results in Chapter 3 are joint work with Tim Roughgarden, and appear in [GR16]. The results in Chapter 4 are joint with Ravi Kumar and Sergei Vassilvitskii, and appear in [GKV16].

Chapter 2

A New Framework for Interaction Networks

Chapter summary: High triangle density, namely the graph property stating that a constant fraction of two-hop paths belong to a triangle, is a common signature of interaction networks. We first study triangle-dense graphs from a structural perspective. We prove constructively that significant portions of a triangle-dense graph are contained in a disjoint union of dense, radius 2 subgraphs. This result quantifies the extent to which triangle-dense graphs resemble unions of cliques. We also show that our algorithm recovers planted clusterings in approximation-stable k-median instances, and empirically show that the algorithm can be an effective "divide" step in a new divide-and-conquer algorithm to approximately count the number of small cliques.

2.1 Introduction

Can the special structure possessed by social and interaction networks be exploited algorithmically? Answering this question requires a formal definition of interaction network structure. Extensive work on this topic has generated countless proposals but little consensus (see e.g. [CF06]). The most oft-mentioned (and arguably most validated) statistical properties of social and interaction networks include heavy-tailed degree distributions [BA99, BKM$^+$00, FFF99], a high density of triangles [WS98, SCW$^+$10, UKBM11] and other dense subgraphs or "communities" [For10, GN02, New03, New06, LLDM08], and low diameter and the small world property [Kle00a, Kle00b, Kle02, New01].

Much of the recent mathematical work on interaction networks has focused on the important goal of developing generative models that produce random networks with many of the above statistical properties. Well-known examples of such models include preferential attachment [BA99] and related copying models [KRR$^+$00], Kronecker graphs [CZF04, LCK$^+$10], and the Chung-Lu random graph model [CL02a, CL02b]. A generative model articulates a hypothesis about what "real-world" interaction networks look like, and is directly useful for generating synthetic data. Once a particular generative model is adopted, a natural goal is to design algorithms tailored to perform well on the instances generated by the model. It can also be used as a proxy to study the effect of random

processes (like edge deletions) on a network. Examples of such results include [AJB00, LAS+08, MS10].

In this chapter, we pursue a different approach. In lieu of adopting a particular generative model, we ask:

> *Is there a combinatorial assumption weak enough to hold in every "reasonable" model of interaction networks, yet strong enough to permit useful structural and algorithmic results?*

Specifically, we seek structural results that apply to *every* reasonable model of interaction networks, including those yet to be devised.

2.1.1 Triangle-Dense Graphs

We initiate the study of *triangle-dense graphs.* Let a *wedge* be a two-hop path in an undirected graph.

Definition 2.1 (Triangle-Dense Graph)**.** *The* triangle density *of an undirected graph $G = (V, E)$ is $\tau(G) := 3t(G)/w(G)$, where $t(G)$ is the number of triangles in G and $w(G)$ is the number of wedges in G (conventionally, $\tau(G) = 0$ if $w(G) = 0$). The class of* ϵ-triangle dense graphs *consists of the graphs G with $\tau(G) \geq \epsilon$.*

Since every triangle of a graph contains 3 wedges, and no two triangles share a wedge, the triangle density of a graph is between 0 and 1. In the social sciences, triangle density is usually called the *transitivity* of a graph [WF94], and also the *(global) clustering coefficient.* We use the term triangle density because "transitivity" already has strong connotations in graph theory.

As an example, the triangle density of a graph is 1 if and only if it is the union of cliques. The triangle density of an Erdös-Renyi graph, drawn from $G(n, p)$, is concentrated around p. Thus, only dense Erdös-Renyi graphs have constant triangle density (as $n \to \infty$). Interaction networks are generally sparse and yet have remarkably high triangle density; the Facebook graph, for instance, has triangle density 0.16 [UKBM11]. Large triangle density — meaning much higher that what the edge density would suggest — is perhaps the least controversial signature of social and interaction networks (see related work below).

The class of ϵ-triangle dense graphs becomes quite diverse as soon as ϵ is bounded below 1. For example, the complete tripartite graph is triangle dense with $\epsilon \approx \frac{1}{2}$. Every graph obtained from a bounded-degree graph by replacing each vertex with a triangle is triangle dense, where ϵ is a constant that depends on the maximum degree. Adding a clique on $n^{1/3}$ vertices to a bounded-degree n-vertex graph produces a triangle-dense graph, where again the constant ϵ depends on the maximum degree. We give a litany of examples in Section 2.4. Can there be interesting structural or algorithmic results for this rich class of graphs?

2.1.2 Main Results

Decomposition Theorems Our main decomposition theorem quantifies the extent to which a graph with large triangle density resembles a union of cliques. The next definition gives our notion of an "approximate union of cliques." We use $G|_S$ to denote the subgraph of a graph G induced by a subset S of vertices. Also, the *edge density* of a graph $G = (V, E)$ is $|E|/\binom{|V|}{2}$. Note that the definition here is slightly different from the formulation in Section 1.1.

Definition 2.2 (Tightly-Knit Family). *Let $\rho > 0$. A collection $V_1, V_2, \ldots, V_k$ of disjoint sets of vertices of a graph $G = (V, E)$ forms a ρ-*tightly-knit family *if:*

- *Each subgraph $G|_{V_i}$ has both edge density and triangle density at least ρ.*
- *Each subgraph $G|_{V_i}$ has radius at most 2.*

When ρ is a constant (as the graph size tends to infinity), we often refer simply to a tightly-knit family. The "clusters" (i.e., the V_i's) of a tightly-knit family are dense in edges and in triangles. In the context of social networks, an abundance of triangles is generally associated with meaningful social structure. There is no a priori restriction on the number of clusters in a tightly-knit family, and the union of the clusters doesn't have to include every vertex of the graph.

Our main decomposition theorem states that every graph with constant triangle density contains a tightly-knit family that captures a constant fraction of the graph's triangles (with the constants depending on the triangle density). The following is proved as Theorem 2.16.

Theorem 2.3 (Main Decomposition Theorem). *There exists a non-constant polynomial $f(\epsilon)$ such that for every ϵ-triangle dense graph G, there exists an $f(\epsilon)$-tightly-knit family that contains an $f(\epsilon)$ fraction of the triangles of G.*

We emphasize that Theorem 2.3 requires only that the input graph G has constant triangle density — beyond this property, it could be sparse or dense, low- or high-diameter, and possess an arbitrary degree distribution. Graphs without constant triangle density, such as sparse Erdös-Renyi random graphs, do not generally admit non-trivial tightly-knit families (even if the triangle density requirement for each cluster is dropped).

Our proof of Theorem 2.3 is constructive. Using suitable data structures, the resulting algorithm can be implemented to run in time proportional to the number of wedges of the graph; a working C++ implementation is available on request. This running time is reasonable for many networks. Our preliminary implementation of the algorithm requires a few minutes on a commodity laptop to decompose networks with millions of edges.

Note that Theorem 2.3 is non-trivial only because we require that the tightly-knit family preserve the "interesting social information" of the original graph, in the form of the graph's triangles. Extracting a single low-diameter cluster rich in edges and triangles is easy — large triangle density implies that typical vertex neighborhoods have these properties. But extracting such a cluster carelessly can do more harm than good, destroying many triangles that only partially intersect the cluster. Our proof of Theorem 2.3 shows how to repeatedly extract low-diameter dense clusters while preserving at least a constant fraction of the triangles of the original graph.

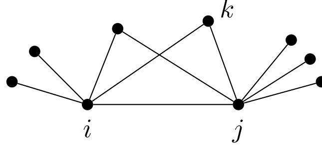

Figure 2.1: Jaccard similarity computation: Edges (i, j) and (j, k) have Jaccard similarity 2/7 and 1/5, respectively. The triangle density of the graph as a whole is 2/9. We use low Jaccard similarity as a signal of low local triangle density.

A graph with constant triangle density need not contain a tightly-knit family that contains a constant fraction of the graph's *edges*; see the examples in Section 2.4. The culprit is that triangle density is a "global" condition and does not guarantee good local triangle density everywhere, allowing room for a large number of edges that are intuitively spurious. Under the stronger condition of constant local triangle density, however, we can compute a tightly-knit family with a stronger guarantee.

Definition 2.4 (Jaccard Similarity)**.** *The* Jaccard similarity *of an edge $e = (i, j)$ of a graph $G = (V, E)$ is the fraction of vertices in the neighborhood of e that participate in triangles:*

$$J_e = \frac{|N(i) \cap N(j)|}{|N(i) \cup N(j) \setminus \{i, j\}|}, \tag{2.1}$$

where $N(x)$ denotes the neighbors of a vertex x in G. If the denominator above is 0, then $J_e = 0$.

The Jaccard similarity is always between 0 and 1. An example computation is in Figure 2.1. Every edge in a clique has Jaccard similarity 1, and a bridge by definition has Jaccard similarity 0. Informally, edges with low Jaccard similarity are often "bridge-like", or are connected to at least one vertex of very high degree. We now introduce a notion of high local triangle density for graphs.

Definition 2.5 (Everywhere Triangle-Dense)**.** *A graph is* everywhere ϵ-triangle dense *if $J_e \geq \epsilon$ for every edge e, and there are no isolated vertices.*

Though useful conceptually, we would not expect graphs in practice to be everywhere triangle dense for a large value of ϵ. The following weaker definition permits graphs that have a small fraction of edges with low Jaccard similarity.

Definition 2.6 (μ, ϵ-Triangle-Dense)**.** *A graph is* μ, ϵ-triangle dense *if $J_e \geq \epsilon$ for at least a μ fraction of the edges e.*

We informally refer to graphs with constant ϵ and high enough μ as *mostly everywhere triangle dense*. An everywhere ϵ-triangle dense graph is μ, ϵ-triangle dense for every μ. An everywhere ϵ-triangle dense graph is also ϵ-triangle dense.

The following is proved as Theorem 2.18.

Theorem 2.7 (Stronger Decomposition Theorem)**.** *There are non-constant polynomials $\mu(\epsilon)$ and $f(\epsilon)$ such that for every $\mu(\epsilon), \epsilon$-triangle dense graph G, there exists an $f(\epsilon)$-tightly-knit family that contains an $f(\epsilon)$-fraction of the triangles* and edges *of G.*

When $\epsilon = 1$, there exists a 1-tightly-knit family that contains all of the triangles and edges of G. A natural question is whether a similarly strong statement holds for ϵ close to 1; e.g. if there exists a tightly-knit family that recovers all but a polynomial in $(1-\epsilon)$ fraction of the triangles or edges. We have the following surprising result for everywhere triangle-dense graphs, proved as Theorem 2.26.

Theorem 2.8 (Sufficiently Triangle-Dense Implies Tightly-Knit). *If G is everywhere $(1-\delta)$-triangle dense for some $\delta \leq 1/3$, then G is the union of components of diameter 2, each of which is $(1-2\delta)^2/(1-\delta)$-edge dense.*

Note that this means G is already a $(1-2\delta)^2/(1-\delta)$-tightly-knit family.

Applications to Planted Cluster Models. We give an algorithmic application of our decomposition in Section 2.6, where the tightly knit family produced by our algorithm is meaningful in its own right. We consider the approximation-stable metric k-median instances introduced by Balcan, Blum, and Gupta [BBG13]. By definition, every solution of an approximation-stable instance that has near-optimal objective function value is structurally similar to the optimal solution. They reduce their problem to clustering a certain graph with "planted" clusters corresponding to the optimal solution. We prove that our algorithm recovers a close approximation to the planted clusters, matching their guarantee.

Applications to Counting Small Cliques. We give a second application of our decomposition algorithm in Section 2.7, to the problem of approximately counting the number of r-cliques in a graph. Here, the tightly-knit family becomes the "divide" step in a divide-and-conquer algorithm. The "conquer" step is a simple sampling algorithm for approximating the number of small cliques in a dense graph, which we use on each subgraph of the tightly-knit family. We run the algorithm on a variety of publicly available interaction networks from the SNAP database [SNA], typically recovering $\sim$90% of the cliques in a very small amount of time.

2.1.3 Model Discussion

Structural Assumptions vs. Generative Models. Pursuing structural results and algorithmic guarantees that assume only a combinatorial condition (namely, constant triangle density), rather than a particular model of interaction networks, has clear advantages and disadvantages. The class of graphs generated by a specific model will generally permit stronger structural and algorithmic guarantees than the class of graphs that share a single statistical property. On the other hand, algorithms and results tailored to a single model can lack robustness: they might not be meaningful if reality differs from the model, and are less likely to translate across different application domains that require different models. Our results for triangle-dense graphs — meaning graphs with constant triangle density — are relevant for every model that generates such graphs with high probability, and we expect that all future interaction network models will have this property. And of course, our results can be used in any application domain that concerns triangle-dense graphs, whether motivated by interaction networks or not.

Beyond generality and robustness, a second reason to prefer a combinatorial assumption to a generative model is that the assumption can be easily verified for a given data set. Since computing the triangle density of a network is a well-studied problem, both theoretically and practically (see [SPK13] and the references therein), the extent to which a network meets the triangle density assumption can be quantified. By contrast, it is not clear how to argue that a network is a typical instance from a generative model, other than by verifying various statistical properties (such as triangle density). This difficulty of verification is amplified when there are multiple generative models vying for prominence, as is currently the case with social and information networks (e.g. [CF06]).

Given the prevalence of triangles in social networks, it is considered an important property for generative models to match [CF06]. Comparative studies of such generative models explicitly compared the clustering coefficients to real data, and showed that classic models like the Preferential Attachment Model, the Copying Model, and the Stochastic Kronecker Model do not generate enough triangles [SCW$^+$10, PSK12]. Recent models have explicitly tried to remedy this by creating many triangles, examples being the Forest Fire, Block Two-Level Erdős-Rényi, and the Transitive Chung-Lu models [LKF07, SKP12, IFMN12].

Why Triangle Density? Interaction networks possess a number of statistical signatures, as discussed above. Why single out triangle density? First, there is tremendous empirical support for large triangle density in interaction networks. This property has been studied for decades in the social sciences [HL70, Col88, Bur04, Fau06, FWVDC10], and recently there have been numerous large-scale studies on online social networks [SCW$^+$10, UKBM11, SPK13]. Second, in light of this empirical evidence, generative models for social and interaction networks are explicitly designed to produce networks with high triangle density [WS98, CF06, SCW$^+$10, VB12]. Third, the assumption of constant triangle density seems to impose more exploitable structure than the other most widely accepted properties of social and interaction networks. For example, the property of having small diameter indicates little about the structure of a network — every network can be rendered small-diameter by adding one extra vertex connected to all other vertices. Similarly, merely assuming a power-law degree distribution does not seem to impose significant restrictions on a graph [FPP06]. For example, the Chung-Lu model [CL02a] generates power-law graphs with no natural decompositions. While constant triangle density is not a strong enough assumption to exclude all "obviously unrealistic graphs," it nevertheless enables non-trivial decomposition results. Finally, we freely admit that imposing one or more combinatorial conditions other than triangle density could lead to equally interesting results, and we welcome future work along such lines. For example, recent work by Ugander, Backstrom, and Kleinberg [UBK13] suggests that constraining the frequencies of additional small subgraphs could produce a refined model of interaction networks.

Why Tightly-Knit Families? We have intentionally highlighted the existence and computation of tightly-knit families in triangle-dense graphs, rather than the (approximate) solution of any particular computational problem on such graphs. Our main structural result quantifies the extent to which we can "visualize" a triangle-dense graph as, approximately, a union of cliques. This is a familiar strategy for understanding restricted graph classes, analogous to using separator theorems to make precise how planar graphs resemble grids [LT79], tree decompositions to quantify how bounded-treewidth graphs resemble trees [RS86], and the regularity lemma to describe how dense

graphs are approximately composed of "random-like" bipartite graphs [Sze78]. Such structural results provide a flexible foundation for future algorithmic applications. We offer a specific application to recovering planted clusterings and leave as future work the design of more applications.

2.2 An Intuitive Overview

We give an intuitive description of our algorithm and proof. Our approach to finding a tightly-knit family is an iterative extraction procedure. We find a single member of the family, remove this set from the graph (called the *extraction*), and repeat. Let us start with an everywhere ϵ-triangle-dense graph G, and try to extract a single set S. It is easy to check that every vertex neighborhood is dense and has many triangles, and would qualify as a set in a tightly-knit family. But for vertex i, there may be many vertices outside $N(i)$ (the neighborhood of i) that form triangles with a single edge contained in $N(i)$. By extracting $N(i)$, we could destroy too many triangles. We give examples in Section 2.4 where such a naïve approach fails.

Here is a simple greedy fix to the procedure. We start by adding $N(i)$ and i to the set S. If any vertex outside $N(i)$ forms many triangles with the edges in $N(i)$, we just add it to S. It is not clear that we solve our problem by adding these vertices to S, since the extraction of S could still destroy many triangles. We prove that by adding at most d_i vertices (where d_i is the degree of i) with the highest number of triangles to $N(i)$, this "destruction" can be bounded. In other words, $G|_S$ will have a high density, obviously has radius 2 (from i), and will contain a constant fraction of the triangles incident to S.

Naturally, we can simply iterate this procedure and hope to get the entire tightly-knit family. But there is a catch. We crucially needed the graph to be *everywhere* ϵ-triangle-dense for the previous argument. After extracting S, this need not hold. We therefore employ a *cleaning* procedure that iteratively removes edges of low Jaccard similarity and produces an everywhere ϵ-triangle-dense graph for the next extraction. This procedure also destroys some triangles, but we can upper bound this number. As an aside, removing low Jaccard similarity edges has been used for sparsifying real-world graphs by Satuluri, Parthasarathy, and Ruan [SPR11].

When the algorithm starts with an arbitrary ϵ-triangle-dense graph G, it first cleans the graph to get an everywhere ϵ-triangle-dense graph. We may lose many edges during the initial cleaning, and this is inevitable, as examples in Section 2.4 show. In the end, this procedure constructs a tightly-knit family containing a constant fraction of the triangles of the original ϵ-triangle-dense graph.

When G is everywhere or mostly everywhere ϵ-triangle-dense, we can ensure that the tightly-knit family contains a constant fraction (depending on ϵ) of the *edges* as well. Our proof is a non-trivial charging argument. By assigning an appropriate weight function to triangles and wedges, we can charge removed edges to removed triangles. This (constructively) proves the existence of a tightly-knit family with a constant fraction of edges and triangles.

2.3 The Decomposition Procedure

In this section we walk through the proof outlined in Section 2.2 above. We first bound the losses from the cleaning procedure in Section 2.3.2. We then show how to extract a member of a tightly-knit family from a cleaned graph in Section 2.3.3. We combine these two procedures in Theorem 2.16 of Section 2.3.4 to obtain a full tightly-knit family from a triangle-dense graph. Finally, Theorem 2.18 of Section 2.3.5 shows that the procedure also preserves a constant fraction of the edges in a mostly everywhere triangle-dense graph.

2.3.1 Preliminaries

We begin with some notation. Consider a graph $G = (V, E)$. We index vertices with $i, j, k, \ldots$, and say vertex i has degree d_i. We repeatedly deal with subgraphs H of G, and use the $\ldots(H)$ notation for the respective quantities in H. So, $t(H)$ denotes the number of triangles in H, $d_i(H)$ denotes the degree of i in H, and so on. Also, if S is a set of vertices, let $G|_S$ denote the induced subgraph on G.

We conclude the preliminaries with a simple lemma on the properties of everywhere ϵ-triangle-dense graphs.

Lemma 2.9 (Local properties of everywhere triangle-dense graphs)**.** *If H is everywhere ϵ-triangle dense, then $d_i \geq \epsilon\, d_j$ for every edge (i, j). Furthermore, $N(i)$ is ϵ-edge dense for every vertex i.*

Proof. If $d_i \geq d_j$ we are done. Otherwise

$$\epsilon \leq J_{(i,j)} = \frac{|N(i) \cap N(j)|}{|(N(i) \setminus \{j\}) \cup (N(j) \setminus \{i\})|} \leq \frac{d_i - 1}{d_j - 1} \leq \frac{d_i}{d_j},$$

as desired. To prove the second statement, let $S = N(i)$. The number of edges in S is at least

$$\frac{1}{2} \sum_{j \in S} |N(i) \cap N(j)| \geq \frac{1}{2} \sum_{j \in S} J_{(i,j)}(d_i - 1) \geq \frac{\epsilon d_i(d_i - 1)}{2} = \epsilon \binom{d_i}{2}.$$

□

2.3.2 Cleaning a Graph

An important ingredient in our constructive proof is a "cleaning" procedure that constructs an everywhere ϵ-triangle-dense graph from a graph H, given an $\epsilon \in (0, 1]$.

The cleaning procedure $clean_\epsilon$**:** Iteratively remove an arbitrary edge with Jaccard similarity less than ϵ, as long as such an edge exists. Finally, remove all isolated vertices.

The output $clean_\epsilon(H)$ is dependent on the order in which edges are removed, but our results hold for an arbitrary removal order. Satuluri et al. [SPR11] use a more nuanced version of cleaning for

graph sparsification of social networks. They provide much empirical evidence that removal of low Jaccard similarity edges does not destroy interesting graph structure, such as its dense subgraphs. Our arguments below may provide some theoretical justification.

Claim 2.10 (Cleaning preserves triangles)**.** *The number of triangles in $clean_\epsilon(H)$ is at least $t(H) - \epsilon w(H)$, where $w(H)$ is the number of wedges in H.*

Proof. The process $clean_\epsilon$ removes a sequence of edges $e_1, e_2, \ldots$. Let W_l and T_l be the set of wedges and triangles that are removed when e_l is removed. Since the Jaccard similarity of e_l *at this stage* is at most ϵ, $|T_l| \leq \epsilon(|W_l| - |T_l|) \leq \epsilon|W_l|$. All the W_l's (and T_l's) are disjoint. Hence, the total number of triangles removed is $\sum_l |T_l| \leq \epsilon \sum_l |W_l| \leq \epsilon w(H)$. □

We get an obvious corollary by noting that $t(H) = \tau(H) \cdot w(H)/3$.

Corollary 2.11. *The graph $clean_\epsilon(H)$ is everywhere ϵ-triangle dense and has at least $(\tau(H)/3 - \epsilon)w(H)$ triangles.*

2.3.3 Extracting a Single Cluster

Suppose we have an everywhere ϵ-triangle dense graph H. We show how to remove a single cluster of a tightly-knit family. Since the entire focus of this subsection is on H, we drop the $\ldots(H)$ notation.

For a set S of vertices, let t_S denote the number of triangles which have at least one vertex in S, and let $t_S^{(I)} = t(H|_S)$ denote the number of triangles which have all three vertices in S (the I is for "internal"). We have the following key definition:

Definition 2.12 (ρ-extractable)**.** *For $\rho \in (0, 1]$, we say that a set of vertices S is ρ-*extractable *if $H|_S$ is ρ-edge dense, ρ-triangle dense, $H|_S$ has radius 2, and $t_S^{(I)} \geq \rho\, t_S$.*

We now define the following *extract* procedure that finds a single extractable cluster in the graph H.

The extraction procedure *extract***:** Let i be a vertex of maximum degree. For every vertex j, let θ_j be the number of triangles incident on j whose other two vertices are in $N(i)$. Let R be the set of d_i vertices with the largest θ_j values. Output $S = \{i\} \cup N(i) \cup R$.

It is not necessary to start with a vertex of maximum degree, but doing so provides a better dependence on ϵ. Also, note that the $\{i\}$ above is just for clarity; a simple argument shows that $i \in R$.

We start with a simple technical lemma.

Lemma 2.13 (Concentration of vectors with low L_1 and high L_2)**.** *Suppose $x_1 \geq x_2 \geq \cdots > 0$ with $\sum x_j \leq \alpha$ and $\sum x_j^2 \geq \beta$. For all indices $r \leq 2\alpha^2/\beta$, $\sum_{j \leq r} x_j^2 \geq \beta^2 r/4\alpha^2$.*

Proof. If $x_{r+1} \geq \beta/2\alpha$, then $\sum_{j \leq r} x_j^2 \geq \beta^2 r/4\alpha^2$ as desired. Otherwise,

$$\sum_{j>r} x_j^2 \leq x_{r+1} \sum_j x_j \leq \beta/2.$$

Hence, $\sum_{j \leq r} x_j^2 = \sum x_j^2 - \sum_{j>r} x_j^2 \geq \beta/2 \geq \beta^2 r/4\alpha^2$, using the bound given for r. □

The main theorem of the section follows.

Theorem 2.14. *Let H be an everywhere ϵ-triangle dense graph. The procedure extract outputs an $\Omega(\epsilon^4)$-extractable set S of vertices. Furthermore, the number of edges in $H|_S$ is an $\Omega(\epsilon)$-fraction of the edges incident to S.*

Proof. Let $\epsilon > 0$, i a vertex of maximum degree, and $N = N(i)$.

We have $|S| \leq 2d_i$. By Lemma 2.9, $H|_N$ has at least $\epsilon\binom{d_i}{2}$ edges, so $H|_S$ is $\Omega(\epsilon)$-edge dense. By the size of S and maximality of d_i, the number of edges in $H|_S$ is an $\Omega(\epsilon)$-fraction of the edges incident to S. It is also easy to see that $H|_S$ has radius 2. It remains to show that $H|_S$ is $\Omega(\epsilon^4)$-triangle dense, and that $t_S^{(I)} = \Omega(\epsilon^4) t_S$.

For any j, let η_j be the number of edges from j to N, and let θ_j be the number of triangles incident on j whose other two vertices are in N. Let $x_j = \sqrt{2\theta_j}$.

Lemma 2.13 tells us that if we can (appropriately) upper bound $\sum_j x_j$ and lower bound $\sum_j x_j^2$, then the sum of the largest few x_j^2's is significant. This implies that $H|_S$ has sufficiently many triangles. Using appropriate parameters, we show that $H|_S$ contains $\Omega(\text{poly}(\epsilon) \cdot d_i^3)$ triangles, as opposed to trivial bounds that are quadratic in d_i.

Claim 2.15. *We have $\sum_j x_j \leq \sum_{k \in N} d_k$, and $\sum_j x_j^2 \geq \frac{\epsilon}{2} \sum_{k \in N} d_k(H|_N)\, d_k$, where $d_k(H|_N)$ is the degree of vertex k within $H|_N$.*

Proof. We first upper bound $\sum_j x_j$:

$$\sum_j x_j \leq \sum_j \sqrt{2\binom{\eta_j}{2}} \leq \sum_j \eta_j = \sum_{k \in N} d_k.$$

The first inequality follows from $\theta_j \leq \binom{\eta_j}{2}$. The last equality is simply stating that the total number of edges to vertices in N is the same as the total number of edges from vertices in N.

Let t_e be the number of triangles that include the edge e. For every $e = (k_1, k_2)$, $t_e \geq J_e \cdot \max(d_{k_1} - 1, d_{k_2} - 1) \geq \epsilon \cdot \max(d_{k_1} - 1, d_{k_2} - 1)$. Since $\epsilon > 0$, each vertex is incident on at least 1 triangle. Hence all degrees are at least 2, and $d_k - 1 \geq d_k/2$ for all k. This means

$$t_e \geq \frac{\epsilon \cdot \max(d_{k_1}, d_{k_2})}{2} \geq \frac{\epsilon(d_{k_1} + d_{k_2})}{4} \qquad \text{for all } e = (k_1, k_2).$$

We can now lower bound $\sum_j x_j^2$. Abusing notation, $e \in H|_N$ refers to an edge in the induced subgraph. We have

$$\sum_j x_j^2 = \sum_j 2\theta_j = \sum_{e \in H|_N} 2t_e \geq \sum_{(k_1,k_2) \in H|_N} \frac{\epsilon}{2}(d_{k_1} + d_{k_2}) = \frac{\epsilon}{2} \sum_{k \in N} d_k(H|_N)\, d_k.$$

The two sides of the second equality are counting (twice) the number of triangles "to" and "from" the edges of N. □

We now use Lemma 2.13 with $\alpha = \sum_{k \in N} d_k$, $\beta = \frac{\epsilon}{2} \sum_{k \in N} d_k(H|_N)\, d_k$, and $r = d_i$. We first check that $r \leq 2\alpha^2/\beta$. Note that $d_i \geq d_k \geq \epsilon d_i$ for all $k \in N$, by Lemma 2.9 and by the maximality of d_i. Hence,

$$\frac{2\alpha^2}{\beta} = \frac{4}{\epsilon} \frac{\left(\sum_{k \in N} d_k\right)^2}{\sum_{k \in N} d_k(H|_N)\, d_k} \geq \frac{4}{\epsilon} \frac{\epsilon d_i |N| \sum_{k \in N} d_k}{d_i \sum_{k \in N} d_k} \geq 4d_i \geq r,$$

as desired. Let R be the set of $r = d_i$ vertices with the highest value of θ_j, or equivalently, with the highest value of x_j^2. By Lemma 2.13, $\sum_{j \in R} x_j^2 \geq \beta^2 r/4\alpha^2$, or $\sum_{j \in R} \theta_j \geq \beta^2 r/8\alpha^2$. We compute

$$\frac{\beta}{\alpha} = \frac{\epsilon}{2} \frac{\sum_{k \in N} d_k(H|_N)\, d_k}{\sum_{k \in N} d_k} \geq \frac{\epsilon}{2} \min_{k \in N} d_k(H|_N) \geq \frac{\epsilon^2 d_i}{4},$$

which gives $\sum_{j \in R} \theta_j \geq \epsilon^4 d_i^3/128$. For the first inequality above, think of the $d_k / \sum d_k$ as the coefficients in a convex combination of $d_k(H|_N)$'s. For the last inequality, $d_k(H|_N) = t_{(i,k)} \geq J_{(i,k)}(d_i - 1) \geq \epsilon d_i/2$ for all $k \in N$.

Recall $S = N \cup R$ and $|S| \leq 2d_i$. We have

$$t_S^{(I)} \geq \frac{\sum_{j \in R} \theta_j}{3} \geq \frac{\epsilon^4 d_i^3}{384},$$

since triangles contained in N get overcounted by a factor of 3. Since both t_S and the number of wedges in S are bounded above by $|S|\binom{d_i}{2} = \Theta(d_i^3)$, $H|_S$ is $\Omega(\epsilon^4)$-triangle dense, and $t_S^{(I)} = \Omega(\epsilon^4) t_S$, as desired. □

2.3.4 Getting the Entire Family in a Triangle-Dense Graph

We start with a ϵ-triangle-dense graph G and explain how to get the desired entire tightly-knit family. Our procedure — called the decomposition procedure — takes as input a parameter ϵ.

The decomposition procedure: Clean the graph with $clean_\epsilon$, and run the procedure *extract* to get a set S_1. Remove S_1 from the graph, run $clean_\epsilon$ again, and *extract* another set S_2. Repeat until the graph is empty. Output the sets $S_1, S_2, \ldots$.

We now prove our main theorem, Theorem 2.3, restated for convenience.

Theorem 2.16. *Consider a τ-triangle dense graph G and $\epsilon \leq \tau/4$. The decomposition procedure outputs an $\Omega(\epsilon^4)$ tightly-knit family with an $\Omega(\epsilon^4)$-fraction of the triangles of G.*

Proof. We are guaranteed by Theorem 2.14 that $G|_{S_i}$ is $\Omega(\epsilon^4)$-edge and $\Omega(\epsilon^4)$-triangle dense and has radius 2. It suffices to prove that an $\Omega(\epsilon^4)$-fraction of the triangles in G are contained in this family.

Consider the triangles that are *not* present in the tightly-knit family. We call these the destroyed triangles. Such triangles fall into two categories: those destroyed in the cleaning phases, and those destroyed when an extractable set is removed. Let C be the triangles destroyed during cleaning, and let D_k be the triangles destroyed in the kth extraction. By the definition of extractable subsets and Theorem 2.14, $t(G|_{S_k}) = \Omega(\epsilon^4|D_k|)$. Note that C, D_k, and the triangles in $G|_{S_k}$ (over all k) partition the total set of triangles. Hence, we get that $\sum_k t(G|_{S_k}) = \Omega(\epsilon^4(t - |C|))$.

We now bound $|C|$. This follows the proof of Claim 2.10. Let $e_1, e_2, \ldots$ be all the edges removed during cleaning phases. Let W_l and T_l be the set of wedges and triangles that are destroyed when e_l is removed. Since the Jaccard similarity of e_l at the time of removal is at most ϵ, $|T_l| \leq \epsilon(|W_l| - |T_l|) \leq \epsilon|W_l|$. All the W_ls (and T_ls) are disjoint. Hence, $|C| = \sum_l |T_l| \leq \epsilon \sum_l |W_l| = \epsilon w = 3\epsilon t/\tau \leq 3t/4$, and $\sum_k t(G|_{S_k}) = \Omega(\epsilon^4 t)$, as desired. □

We also give a quick runtime analysis. Recall that $w(G)$ is the number of wedges in G. Note that $w(G)$ will be somewhere between the number of edges in G times the average degree of G, and the number of edges times the max degree. For most practical graphs, very few of the edges from high degree vertices will play a role in the clustering. One can also imagine doing a preliminary cleaning step where we first remove all edges (i, j) where $d_i < \epsilon d_j$ (in time proportional to the number of edges), which should reduce the max degree to roughly the size of the biggest cluster.

Theorem 2.17. *The decomposition can be obtained in time proportional to $w(G) + |V|$.*[1]

Proof. We maintain five hash tables/sets for the course of the algorithm: a hash table from each node to its incident edges, a hash table from each edge to the degrees of its endpoints, a hash table from each edge e to the set of triangles containing e, a hash table from integers d to the set of all vertices of degree d, and the set of all edges with Jaccard similarity less than ϵ. We also keep track of the maximum degree.

We assume constant-time insert, delete, and lookup in all hash tables and sets. We can initialize the data structures by enumerating over all wedges and edges, which takes $O(w(G) + |V|)$ time. When an edge e is deleted, we can update the hash tables in time proportional to the number of wedges containing e. Since edges are deleted but never added by the decomposition procedure, we spend a total of $O(w(G) + |V|)$ time maintaining the data structures over the course of the procedure.

Now, the three operations performed by the procedure are *clean*$_\epsilon$, *extract*, and removing a set of nodes from the graph. The first and third are accounted for by our data structures, as is finding a vertex of maximum degree from which to *extract*.

For an *extract* operation from vertex i, we enumerate over all pairs (u, v) of neighbors of i, which takes time proportional to the number of wedges at i. For each pair that is an edge, we enumerate

[1]Note that the algorithm here uses $O(w(G) + |V|)$ space as well. A slight variant of the algorithm runs in $O(w(G) + |V|)$ time and only $O(|E|)$ space, but takes longer to analyze — see Appendix 2.A details.

over all triangles that involve this pair/edge, and by hashing appropriately, we can find the d_i vertices with the largest θ_j values. Every such enumerated triangle is deleted when the extracted set is removed, so the total time spent here is again at most $O(w(G)+|E|) = O(w(G)+|V|)$, giving the desired result. □

2.3.5 Preserving Edges in a Mostly Everywhere Triangle-Dense Graph

For a mostly everywhere triangle-dense graph, the decomposition procedure can also preserve a constant fraction of the *edges*. This requires a more subtle argument. The aim of this subsection is to prove the following (cf. Theorem 2.7).

Theorem 2.18. *Consider a μ, γ-triangle dense graph G, for $\mu \geq 1 - \gamma^2/32$. The decomposition procedure, with $\epsilon \leq \gamma^3/12$, outputs an $\Omega(\epsilon^4)$ tightly-knit family with an $\Omega(\epsilon^4)$ fraction of the triangles of G and an $\Omega(\epsilon\gamma)$ fraction of the edges of G.*

The proof appears at the end of the subsection. The tightly-knit family and triangle conditions follow directly from Theorem 2.16, so we focus on the edge condition. By Theorem 2.14, the actual removal of the clusters preserves a large enough fraction of the edges. The difficulty is in bounding the edge removals during the cleaning phases.

We first give an informal description of the argument. We would like to charge lost edges to lost triangles, and piggyback on the fact that not many triangles are lost during cleaning. More specifically, we apply a weight function to triangles (and wedges), such that losing or keeping an edge corresponds to losing or keeping roughly one unit of triangle (and wedge) weight in the graph. Most edges (i, j) belong to roughly $d_i + d_j$ triangles and wedges, and so intuitively we weight each of those triangles (and wedges) by roughly $1/(d_i + d_j)$. This intuition breaks down if $d_i \ll d_j$, but $d_i \approx d_j$ for edges with high Jaccard similarity.

The rest of the argument follows the high-level plan of the ϵ-triangle dense case (cf. the argument to bound $|C|$ in Theorem 2.16), though work is needed to replace triangles and wedges with their weighted counterparts. The original graph G has high triangle density, which under our weight function is enough to imply a comparable amount of triangle weight and wedge weight. Only edges with low Jaccard similarity are removed during cleaning, and each of these removed edges destroys significantly more wedge weight than triangle weight. Hence, at the end of the process, a lot of triangle weight must remain. There is a tight correspondence between edges and triangle weight, and so a lot of edges must also remain.

We now start the formal proof. We use E, W, and T to denote the sets of edges, wedges, and triangles in G. W_e and T_e denote the sets of edges and triangles that include the edge e. We use E^c, W^c, and T^c to denote the respective sets destroyed during the cleaning phases, and use W_e^c and T_e^c to denote the corresponding local versions. If an edge e is removed during cleaning, then $W_e^c \subseteq W_e$, but the sets are not necessarily equal, since elements of W_e may have been removed prior to e being cleaned. Let $T^s = T \setminus T^c$. Let E^s and V^s denote the edges and vertices, respectively, included in at least one triangle of T^s. For ease of reading, let $d_i' = d_i - 1$ be one less than the degree of vertex i.

Call an edge e *good* if $J_e \geq \gamma$ in the original graph G, and *bad* otherwise. We use g_i to denote the number of good edges incident to vertex i. Call a wedge good if it contains at least one good edge, and bad otherwise. By hypothesis, a μ fraction of edges are good. We make the following observation.

Claim 2.19. *For every good edge $e = (i,j)$, $d'_i \geq \gamma d'_j$.*

Proof. We have

$$\gamma \leq J_e = \frac{t_e}{d'_i + d'_j - t_e} \leq \frac{d'_i}{d'_j},$$

where the last inequality comes from $t_e \leq d'_i$. ☐

We now define a *weight* function r on triangles and wedges, as per the informal argument above. For a triangle $\mathcal{T} = (i_1, i_2, i_3)$ with at least 2 good edges, let $r(\mathcal{T}) = 1/d'_{i_1} + 1/d'_{i_2} + 1/d'_{i_3}$. If $\mathcal{T}$ has only one good edge (i_1, i_2), let $r(\mathcal{T}) = 1/d'_{i_1} + 1/d'_{i_2}$. If $\mathcal{T}$ has no good edges, let $r(\mathcal{T}) = 0$. For a good wedge w with central vertex i, let $r(w) = 1/d'_i$, otherwise, let $r(w) = 0$. Let $r(X) = \sum_{x \in X} r(x)$. Note that weights are always with respect to the degrees in the original graph G, and do not change over time.

In the next two claims we show that the total triangle weight in G is comparable to the total wedge weight in G, and is also comparable to $|E|$.

Claim 2.20. $r(T) \geq \gamma\mu|E|$.

Proof. Let t_i^g be the number of triangles $(i,j,k) \in T$ for which at least one of $(i,j),(i,k)$ is good. Since the good edges each have Jaccard similarity $\geq \gamma$, we have $t_i^g \geq g_i \gamma d'_i/2$. Thus,

$$r(T) = \sum_i \frac{t_i^g}{d'_i} \geq \sum_i \frac{g_i \gamma}{2} = \gamma\mu|E|.$$ ☐

Claim 2.21. $r(W) \leq 2\mu|E|$.

Proof. Let w_i^g be the number of good wedges which have i as their central vertex. Then

$$r(W) = \sum_i \frac{w_i^g}{d'_i} \leq \sum_i g_i = 2\mu|E|.$$ ☐

The next two claims bound the triangle weight lost by cleaning any particular edge.

Claim 2.22. *If a good edge $e = (i,j)$ is removed during cleaning, then $r(T_e^c) \leq (3\epsilon/\gamma^2) r(W_e^c)$.*

Proof. Assume that $d_i \geq d_j$. Let $d = d_i$. We first lower bound $r(W_e^c)$ as a function of $|W_e^c|$. For any $w \in W_e^c$, w has at least one good edge, and has either i or j as its central vertex. Hence $r(w) \geq \min\{1/d'_i, 1/d'_j\} = 1/d'$, and

$$r(W_e^c) \geq \frac{|W_e^c|}{d'}.$$

We now upper bound $r(T_e^c)$ as a function of $|T_e^c|$. Consider triangle $t = (i, j, k) \in T_e^c$. If (i, j) is the only good edge in t, then $r(t) = 1/d'_i + 1/d'_j \leq 2/d'\gamma$, since $d'_j \geq d'\gamma$ by Claim 2.19. If t has at least 2 good edges, then k is at most 2 good edges away from i, and $d'_k \geq d'\gamma^2$. This gives $r(t) = 1/d'_i + 1/d'_j + 1/d'_k \leq 3/d'\gamma^2$. Hence

$$r(T_e^c) \leq \max\left\{\frac{3}{d'\gamma^2}, \frac{2}{d'\gamma}\right\} |T_e^c| = \frac{3}{d'\gamma^2}|T_e^c|.$$

Now, $|T_e^c| \leq \epsilon|W_e^c|$, since $J_e \leq \epsilon$ at the time of cleaning. Hence we have

$$r(T_e^c) \leq \frac{3}{d'\gamma^2}|T_e^c| \leq \frac{3\epsilon}{d'\gamma^2}|W_e^c| \leq \frac{3\epsilon}{\gamma^2}\, r(W_e^c)$$

as desired. □

Claim 2.23. *If a bad edge $e = (i, j)$ is removed during cleaning, $r(T_e^c) \leq 4/\gamma$.*

Proof. The only triangles with non-zero weight in T_e^c have a good edge to i and/or a good edge to j. Let m_i and m_j be the minimum degrees of any vertex connected by a good edge to i and j, respectively. It is not too hard to see that

$$r(T_e^c) \leq g_i\left(\frac{1}{d'_i} + \frac{1}{m'_i}\right) + g_j\left(\frac{1}{d'_j} + \frac{1}{m'_j}\right),$$

since there are at most $g_i + g_j$ triangles with non-zero weight incident to e, each with at most the weight in the corresponding set of parentheses. Plugging in $m'_i \geq \gamma d'_i$ (Claim 2.19) and $g_i \leq d'_i$ gives the desired result. □

We now combine the observations above to show that cleaning cannot remove all the triangle weight.

Claim 2.24. $r(T^s) \geq \gamma|E|/4$.

Proof. We have

$$\begin{aligned}
r(T^c) &= \sum_{\text{good } e} r(T_e^c) + \sum_{\text{bad } e} r(T_e^c) \\
&\leq \sum_{\text{good } e} \frac{3\epsilon}{\gamma^2} r(W_e^c) + \sum_{\text{bad } e} \frac{4}{\gamma} && \text{by Claim 2.22 and Claim 2.23} \\
&\leq \frac{3\epsilon}{\gamma^2} r(W) + \frac{4}{\gamma}(1-\mu)|E| \\
&\leq \frac{6\epsilon\mu|E|}{\gamma^2} + \frac{4(1-\mu)|E|}{\gamma} && \text{by Claim 2.21} \\
&\leq \frac{\gamma\mu|E|}{2} + \frac{\gamma|E|}{8},
\end{aligned}$$

where the last inequality follows from the bounds on ϵ and μ in the theorem statement. Hence

$$
\begin{aligned}
r(T^s) &= r(T) - r(T^c) \\
&\geq \gamma\mu|E| - \left(\frac{\gamma\mu|E|}{2} + \frac{\gamma|E|}{8}\right) && \text{by Claim 2.20} \\
&\geq \gamma|E|/4,
\end{aligned}
$$

since $\mu \geq 3/4$. $\square$

Finally, we show that if a subgraph of G has high triangle weight, it must also have a lot of edges. Though the claim is stated in terms of T^s, the proof would hold for any $H \subset G$. This can be thought of as a moral converse to Claim 2.20.

Claim 2.25. $r(T^s) \leq |E^s|$.

Proof. Let $H = (V, E^s)$. The triangles of H are exactly T^s. We have

$$
r(T^s) \leq \sum_{(i,j,k) \in T(H)} \frac{1}{d'_i(G)} + \frac{1}{d'_j(G)} + \frac{1}{d'_k(G)} = \sum_i \frac{t_i(H)}{d'_i(G)},
$$

where $t_i(H)$ is the number of triangles in H incident to i. From here, we compute

$$
\sum_i \frac{t_i(H)}{d'_i(G)} \leq \frac{\binom{d_i(H)}{2}}{d'_i(G)} \leq \sum \frac{d_i(H)}{2} = |E^s|
$$

as desired. $\square$

The last two claims together imply that the cleaning phase does not destroy too many edges. The rest of the proof is nearly identical to that of Theorem 2.16 from the ϵ-triangle dense case.

Proof. (of Theorem 2.18) As noted above, the tightly-knit family and triangle conditions follow directly from Theorem 2.16.

Let D_k be the edges destroyed in the kth extraction, and let E_k be the edges in $G|_{S_k}$. By Theorem 2.14, $|E_k| = \Omega(\epsilon|D_k|)$. Since E^c, D_k, and E_k (over all k) partition E, we have $\sum_k |E_k| = \Omega(\epsilon(|E| - |E^c|))$. Since $|E^c| + |E^s| \leq |E|$, we have $\sum_k |E_k| = \Omega(\epsilon|E^s|)$. Finally, by Claim 2.24 and Claim 2.25, $|E^s| = \Omega(\gamma|E|)$, and so $\sum_k |E_k| = \Omega(\epsilon\gamma|E|)$ as desired. $\square$

2.4 Triangle-Dense Graphs: The Rogues' Gallery

This section provides a number of examples of graphs with constant triangle density. These examples show, in particular, that radius-1 clusters are not sufficient to capture a constant fraction of an ϵ-triangle-dense graph's triangles, and that tightly knit families cannot always capture a constant fraction of an ϵ-triangle-dense graph's edges.

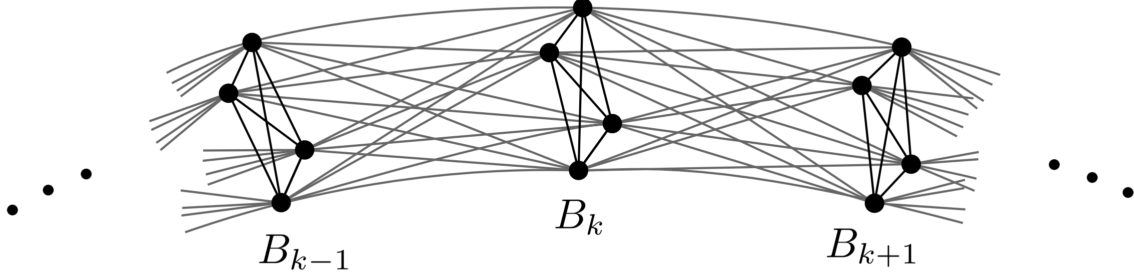

Figure 2.2: Bracelet graph with $d/3 = 4$: a triangle-dense graph that is far from a union of cliques.

• **Why radius 2?** Consider the complete tripartite graph. This is everywhere ϵ-triangle-dense with $\epsilon \approx \frac{1}{2}$. If we removed the 1-hop neighborhood of any vertex, we would destroy a $1 - \Theta(1/n)$-fraction of the triangles. The only tightly-knit family (with constant ρ) in this graph is the entire graph itself.

• **More on 1-hop neighborhoods.** All 1-hop neighborhoods in an everywhere triangle-dense graph are edge-dense, in the sense of Lemma 2.9. Maybe we could just take the 1-hop neighborhoods of an independent set, to get a tightly-knit family? Of course, the clusters would only be edge-disjoint (not vertex disjoint).

We construct a family of everywhere ϵ-triangle-dense graph with constant ϵ where this does not work. There are $m + 1$ disjoint sets of vertices, $A_1, \ldots, A_m, B$ each of size m. The graph induced on $\cup_k A_k$ is just a clique on m^2 vertices. Each vertex $b_k \in B$ is connected to all of A_k. Note that B is a maximal independent set, and the 1-hop neighborhoods of B contain $\Theta(m^4)$ triangles in total. However, the total number of triangles in the graph is $\Theta(m^6)$.

• **Why we can't preserve edges.** Theorem 2.3 only guarantees that the tightly-knit family contains a constant fraction of the graph's triangles, not its edges. Consider a graph that has a clique on $n^{1/3}$ vertices and an arbitrary (or say, a random) constant-degree graph on the remaining $n - n^{1/3}$ vertices. No tightly-knit family (with constant ρ) can involve vertices outside the clique, so most of the edges must be removed. Of course, most edges in this case have low Jaccard similarity.

In general, the condition of constant triangle density is fairly weak and is met by a wide variety of graphs. The following two examples provide further intuition for this class of graphs.

• **A family of triangle-dense graph far from a disjoint union of cliques.** Define the graph Bracelet(m, d), for m nodes of degree d, when $m > 4d/3$, as follows: Let $B_1, \ldots, B_{3m/d}$ be sets of $d/3$ vertices each put in cyclic order. Note that $3m/d \geq 4$. Connect each vertex in B_k to each vertex in B_{k-1}, B_k and B_{k+1}. Refer to Figure 2.2. This is an everywhere ϵ triangle-dense d-regular graph on m vertices, with ϵ a constant as $m \to \infty$. Nonetheless, it is maximally far (i.e., $O(md)$ edges away) from a disjoint union of cliques. A tightly-knit family is obtained by taking $B_1 \cup B_2 \cup B_3$, $B_4 \cup B_5 \cup B_6$, etc.

• **Hiding a tightly-knit family.** Start with $n/3$ disjoint triangles. Now, add an arbitrary bounded-degree graph (say, an expander) on these n vertices. The resulting graph has constant triangle density, but most of the structure is irrelevant for a tightly-knit family.

2.5 Recovering Everything in an Everywhere 2/3-Triangle-Dense Graph

A natural question is what happens to ϵ-triangle-dense graphs, as ϵ goes to 1. We have the following surprising result for everywhere triangle-dense graphs:

Theorem 2.26. *If G is everywhere $(1-\delta)$-triangle dense for some $\delta \le 1/3$, then G is the union of components of diameter 2, each of which is $(1-2\delta)^2/(1-\delta)$-edge dense.*

Note that this implies that our decomposition algorithm will recover the entire graph.

Proof. We repeatedly use the following identity: for sets X and Y, $X \cup Y = X + Y - X \cap Y$. For any sets I, J, K, and L, we have

$$\begin{aligned} I \cup J \cup K \cup L &= I \cup J \cup K + L - (I \cup J \cup K) \cap L \\ &\le I \cup J \cup K + L - K \cap L \\ &= I \cup J + K - (I \cup J) \cup K + L - K \cap L \\ &\le I \cup J + K - J \cup K + L - K \cap L \\ &= I + J + K + L - I \cap J - J \cap K - K \cap L. \end{aligned}$$

Let (i,j), (j,k), and (k,l) be edges in G. Say for contradiction the distance from i to l is 3. Let $N_i = N(i) \setminus \{j\}, N_j = N(j) \setminus \{i,k\}, N_k = N(k) \setminus \{j,l\}$, and $N_l = N(l) \setminus \{k\}$. Note that $N(i) \cap N(j) = N_i \cap N_j$, and $N(i) \cup N(j) - 2 \ge N_i \cup N_j \ge N_j$, so

$$\frac{2}{3} \le J_{(i,j)} = \frac{N(i) \cap N(j)}{N(i) \cup N(j) - 2} \le \frac{N_i \cap N_j}{N_j}.$$

Similarly, $2/3 \le (N_k \cap N_l)/N_k$. We have a strict inequality for the middle edge: $N(j) \cup N(k) - 2 > \max(N_j, N_k) \ge (N_j + N_k)/2$, and so $1/3 < (N_j \cap N_k)/(N_j + N_k)$.

From there,

$$\begin{aligned} N_i \cup N_l &\le N_i \cup N_j \cup N_k \cup N_l \\ &\le N_i + N_j + N_k + N_l - N_i \cap N_j - N_j \cap N_k - N_k \cap N_l \\ &< N_i + N_j + N_k + N_l - 2N_j/3 - (N_j + N_k)/3 - 2N_k/3 \\ &= N_i + N_l. \end{aligned}$$

Hence $N_i \cap N_l$ is non-empty, and there is a path of length 2 from i to l, contradicting our original claim. This demonstrates that every component of G has diameter 2.

Now let C be a component of G, and let i be a vertex of maximal degree d_i in C. Let $k \in C \setminus N(i)$, and let $j \in N(i) \cap N(k)$. Let $N_i = N(i) \setminus \{j\}$ and so on, as above.

Then $N_i \cap N_k = N_i + N_k - N_i \cup N_k \ge N_i + N_k - N_i \cup N_j \cup N_k \ge -N_j + N_i \cap N_j + N_j \cap N_k$, by the same argument as above. We also similarly have $N_i \cap N_j \ge (1-2\delta)N_i + \delta N_j$ and $N_j \cap N_k \ge (1-\delta)N_j$,

which implies $N_i \cap N_k \geq (1-2\delta)N_i = (1-2\delta)(d_i - 1)$. Adding j back in, we have

$$N(i) \cap N(k) \geq (1-2\delta)d_i.$$

Now, pick a vertex $l \in N(i)$. Let $N_l = N(l) \setminus \{i\}$, and $N_i = N(i) \setminus \{l\}$. We have $N_l \setminus N_i = N_l - N_l \cap N_i \leq N_l - (1-\delta)N_i \leq \delta N_i = \delta(d_i - 1) < \delta d_i$, where the second inequality holds because d_i is maximal.

Putting this together, the total number of edges leaving $N^*(i) = i \cup N(i)$ is at most $N(i)\delta d_i = \delta d_i^2$. Each vertex in $C \setminus N^*(i)$ has at least $(1-2\delta)d_i$ edges entering $N^*(i)$. Hence there are at most $\delta d_i/(1-2\delta)$ vertices in $C \setminus N^*(i)$, and at most $\delta d_i/(1-2\delta) + d_i + 1$ vertices in C.

Finally, each vertex in C has degree at least $(1-2\delta)d_i$, and so the edge density in C is at least $\frac{(1-2\delta)d_i}{\delta d_i/(1-2\delta)+d_i} = \frac{(1-2\delta)^2}{1-\delta}$ as desired. □

It is worth noting that the bracelet graph from Figure 2.2 is just under everywhere 1/2-triangle dense, has arbitrarily high diameter, and has arbitrarily low edge density. We conjecture that regular graphs that are just above everywhere 1/2-triangle dense have diameter 2 by the same argument as above.

2.6 Recovering a Planted Clustering

This section gives an algorithmic application of our decomposition procedure to recovering a "ground truth" clustering. We study the planted clustering model defined by Balcan, Blum, and Gupta [BBG13], and show that our algorithm gives guarantees similar to theirs. We do not subsume the results in [BBG13]. Rather, we observe that a graph problem that arises as a subroutine in their algorithm is essentially that of finding a tightly-knit family in a triangle-dense graph. Their assumptions ensure that there is (up to minor perturbations) a unique such family.

The main setting of [BBG13] is as follows. Given a set of points V in some metric space, we wish to k-cluster them to minimize some fixed objective function, such as the k-median objective. Denote the optimal k-clustering by $\mathcal{C}$ and the value by OPT. The instance satisfies (c, ϵ)-*approximation-stability* if for any k-clustering $\mathcal{C}'$ of V with objective function value at most $c \cdot OPT$, the "classification distance" (formalized as Δ-incorrect below) between $\mathcal{C}$ and $\mathcal{C}'$ is at most ϵ. Thus, all solutions with near-optimal objective function value must be structurally close to $\mathcal{C}$.

A summary of the argument in [BBG13] is as follows. The first step converts an approximation-stable k-median instance into an unweighted undirected graph by including an edge between two points if and only if the distance between them (in the k-median instance) is at most a judiciously chosen threshold τ. In [BBG13, Lemma 3.5] it is proved that this *threshold graph* $G = (V, E)$ contains k disjoint cliques $\{X_a\}_{a=1}^k$, such that the cliques do not have any common neighbors. These cliques correspond to clusters in the ground-truth clustering, and their existence is a consequence of the approximation stability assumption. The aim is to get a k-clustering sufficiently close to $\{X_a\}$. Formally, a k-clustering $\{S_a\}$ of V is Δ-*incorrect* if there is a permutation σ such that $\sum |X_{\sigma(a)} \setminus S_a| \leq \Delta$.

Let $B = V \setminus \bigcup_a X_a$. The second step of the argument in [BBG13] proves that when $|B|$ is small, good approximations to $\{X_a\}$ can be found efficiently. We give a different algorithm for implementing this second step; correctness follows by adapting the arguments in [BBG13]. Intuitively, the connection between our work and the setting of [BBG13] is that, when $|B|$ is much smaller than $\sum_a |X_a|$, the threshold graph G output by the first step has high triangle density. Furthermore, as we prove below, the clusters output by the procedure *extract* of Theorem 2.14 are very close to the X_a's of the threshold graph.

In more detail, to obtain a k-clustering of the threshold graph G identified in [BBG13], our algorithm iteratively use the procedure *extract* (from Section 2.3.3) k times to get clusters $S_1, S_2, \ldots, S_k$. In particular, recall that at each step we choose a vertex s_i with the current highest degree d_i. We set N_i to be the d_i neighbors of s_i at this time, and R to be the d_i vertices with the largest number of triangles to N_i. Then, $S_i = \{i\} \cup N_i \cup R$. The exact procedure of Theorem 2.16, which includes cleaning, also works fine. Foregoing the cleaning step does necessitate a small technical change to *extract*: instead of adding all of R to S, we only add the elements of R which have a positive number of triangles to N_i.

We use the notation $N^*(U) = N(U) \cup U$. So $N^*(X_a) \cap N^*(X_b) = \emptyset$, when $a \neq b$. Unlike [BBG13], we assume that $|X_a| \geq 3$. The following parallels the main theorem of [BBG13, Theorem 3.9], and the proof has the same high-level structure.

Theorem 2.27. *The output of the clustering algorithm above is $O(|B|)$-incorrect on G.*

Proof. We first map the algorithm's clustering to the true clustering $\{X_a\}$. Our algorithm outputs k clusters, each with an associated "center" (the starting vertex). These are denoted $S_1, S_2, \ldots$, with centers $s_1, s_2, \ldots$, in order of extraction. We determine if there exists some true cluster X_a, such that $s_1 \in N^*(X_a)$. If so, we map S_1 to X_a. (Recall the $N^*(X_a)$'s are disjoint, so X_a is unique if it exists.) If no X_a exists, we simply do not map S_1. We then perform this for $S_2, S_3, \ldots$, except that we do not map S_k if we would be mapping it to an X_a that has previously been mapped to. We finally end up with a subset $P \subseteq [k]$, such that for each $a \in P$, S_a is mapped to some $X_{a'}$. By relabeling the true clustering, we can assume that for all $a \in P$, S_a is mapped to X_a. The remaining clusters (for $X_{a \notin P}$) can be labeled with an arbitrary permutation of $[k] \setminus P$.

Our aim is to bound $\sum_a |X_a \setminus S_a|$ by $O(|B|)$.

We perform some simple manipulations.

$$\begin{aligned}
\bigcup_a (X_a \setminus S_a) &= \bigcup_{a \in P} (X_a \setminus S_a) \cup \bigcup_{a \notin P} (X_a \setminus S_a) \\
&= \bigcup_{a \in P} (X_a \cap \bigcup_{b<a} S_b) \cup \bigcup_{a \in P} (X_a \setminus \bigcup_{b \leq a} S_b) \cup \bigcup_{a \notin P} (X_a \setminus S_a) \\
&\subseteq \bigcup_{a} (X_a \cap \bigcup_{b<a} S_b) \cup \bigcup_{a \in P} (X_a \setminus \bigcup_{b \leq a} S_b) \cup \bigcup_{a \notin P} X_a.
\end{aligned}$$

So we get the following sets of interest.

• $L_1 = \bigcup_a (X_a \cap \bigcup_{b<a} S_b) = \bigcup_b (S_b \cap \bigcup_{a>b} X_a)$ is the set of vertices that are "stolen" by clusters before S_a.

• $L_2 = \bigcup_{a \in P} (X_a \setminus \bigcup_{b \le a} S_b)$ is the set of vertices that are left behind when S_a is created.

• $L_3 = \bigcup_{a \notin P} X_a$ is the set of vertices that are never clustered.

Note that $\sum_a |X_a \setminus S_a| = |\bigcup_a (X_a \setminus S_a)| \le |L_1| + |L_2| + |L_3|$. The proof is completed by showing that $|L_1| + |L_2| + |L_3| = O(|B|)$. This will be done through a series of claims.

We first state a useful fact.

Claim 2.28. *Suppose for some $b \in \{1, 2, \ldots, k\}$, $s_b \in N(X_b)$. Then N_b is partitioned into $N_b \cap X_b$ and $N_b \cap B$.*

Proof. Any vertex in $N_b \setminus X_b$ must be in B. This is because N_b is contained in a two-hop neighborhood from X_b, which cannot intersect any other X_a. □

Claim 2.29. *For any b, $|S_b \cap \bigcup_{a>b} X_a| \le 6|S_b \cap B|$.*

Proof. We split into three cases. For convenience, let U be the set of vertices $S_b \cap \bigcup_{a>b} X_a$. Recall that $|S_b| \le 2d_b$.

• For some c, $s_b \in X_c$: Note that $c \le b$ by the relabeling of clusters. Observe that S_b is contained in a two-hop neighborhood of s_b, and hence cannot intersect any cluster X_a for $a \ne c$. Hence, U is empty.

• For some (unique) c, $s_b \in N(X_c)$: Again, $c \le b$. By Claim 2.28, $d_b = |N_b| = |N_b \cap X_c| + |N_b \cap B|$. Suppose $|N_b \cap B| \ge d_b/3$. Then $|S_b \cap B| \ge |N_b \cap B| \ge d_b/3$. We can easily bound $|S_b| \le 2d_b \le 6|S_b \cap B|$.

Suppose instead $|N_b \cap B| < d_b/3$, and hence $|N_b \cap X_c| > 2d_b/3$. Note that $|N_b \cap X_c|$ is a clique. Each vertex in $N_b \cap X_c$ makes $\binom{|N_b \cap X_c|-1}{2} \ge \binom{\lfloor 2d_b/3 \rfloor}{2}$ triangles in N_b. On the other hand, the only vertices of N_b that any vertex in X_a for $a \ne c$ can connect to is in $N_b \cap B$. This forms fewer than $\binom{\lfloor d_b/3 \rfloor}{2}$ triangles in N_b. If $\binom{\lfloor d_b/3 \rfloor}{2} > 0$, then $\binom{\lfloor 2d_b/3 \rfloor}{2} > \binom{\lfloor d_b/3 \rfloor}{2}$.

Consider the construction of S_b. We take the top d_b vertices with the most triangles to N_b, and say we insert them in decreasing order of this number. Note that in the modified version of the algorithm, we only insert them while this number is positive. Before any vertex of X_a ($a \ne b$) is added, all vertices of $N_b \cap X_c$ must be added. Hence, at most $d_b - |N_b \cap X_c| = |N_b \cap B| \le |S_b \cap B|$ vertices of $\cup_{a \ne b} X_a$ can be added to S_b. Therefore, $|U| \le |S_b \cap B|$.

• The vertex s_b is at least distance 2 from every X_c: Note that $N_b \subseteq S_b \cap B$. Hence, $|S_b| \le 2d_b \le 2|S_b \cap B|$. □

Claim 2.30. *For any $a \in P$, $|X_a \setminus \bigcup_{b \leq a} S_b| \leq |S_a \cap B|$.*

Proof. Since $a \in P$, either $s_a \in X_a$ or $s_a \in N(X_a)$. Consider the situation of the algorithm after the first $a-1$ sets $S_1, S_2, \ldots, S_{a-1}$ are removed. There is some subset of X_a that remains; call it $X'_a = X_a \setminus \bigcup_{b<a} S_b$.

Suppose $s_a \in X_a$. Since X'_a is still a clique, $X'_a \subseteq N_a$, and $(X_a \setminus \bigcup_{b \leq a} S_b)$ is empty.

Suppose instead $s_a \in N(X_a)$. Because s_a has maximum degree and X'_a is a clique, $d_a \geq |X'_a| - 1$. Note that $|X'_a \setminus S_a|$ is what we wish to bound, and $|X'_a \setminus S_a| \leq |X'_a \setminus N_a|$. By Claim 2.28, N_a partitions into $N_a \cap X_a = N_a \cap X'_a$ and $N_a \cap B$. We have $|X'_a \setminus N_a| = |X'_a| - |N_a \cap X_a| \leq d_a + 1 - |N_a \cap X_a| = |N_a \cap B| + 1 \leq |S_a \cap B|$. □

Claim 2.31. $|L_3| \leq |B| + |L_1|$.

Proof. Consider some X_a for $a \notin P$. Look at the situation when $S_1, \ldots, S_{a-1}$ are removed. There is a subset X'_a (forming a clique) left in the graph. All the vertices in $X_a \setminus X'_a$ are contained in L_1. By maximality of degree, $d_a \geq |X'_a| - 1$. Furthermore, since $a \notin P$, $N_a \subseteq B$ implying $d_a \leq |S_a \cap B| - 1$. Therefore, $|X'_a| \leq |S_a \cap B|$. We can bound $\bigcup_{a \notin P} (X_a \setminus X'_a) \subseteq L_1$, and $\sum_{a \notin P} |X'_a| \leq |B|$, completing the proof. □

To put it all together, we sum the bound of Claim 2.29 and Claim 2.30 over $b \in [k]$ and $a \in P$ respectively to get $|L_1| \leq 6|B|$ and $|L_2| \leq |B|$. Claim 2.31 with the bound on $|L_1|$ yields $|L_3| \leq 7|B|$, completing the proof of Theorem 2.27. □

2.7 Experiments in Counting Small Cliques

This section presents a practical application of the decomposition procedure. In particular, we use the decomposition procedure to give one of the first scalable algorithms for approximately counting the number of small cliques in triangle-dense networks.

Counting the number of r-cliques is a fundamental operation in graph analysis. These numbers are an important part of graphlet analysis in bioinformatics [PCJ04, HBPS07, GK07, ADH$^+$08], subgraph frequency counts in social network analysis [IWM00, YH02, KK04, RBH12, UBK13], and are inputs to graph models (like exponential random graph models [CJ12]). The simplest non-trivial incarnation of this problem is *triangle counting*, with $r = 3$. Triangle counting has a long and rich history in data mining research [Tso08, TKMF09, CC11, SPK13], but there has been little work done for larger values of r.

We provide one of the first scalable algorithms, $\mathsf{CES}(r)$ (for "clean-extract-sample"), for approximately counting r-cliques in triangle-dense networks. We first partition the graph using the decomposition algorithm of Section 2.3, and then approximately count the r-cliques in each subgraph using sampling methods. Our sampling procedures explicitly exploit the density of the subgraphs output by the decomposition algorithm.

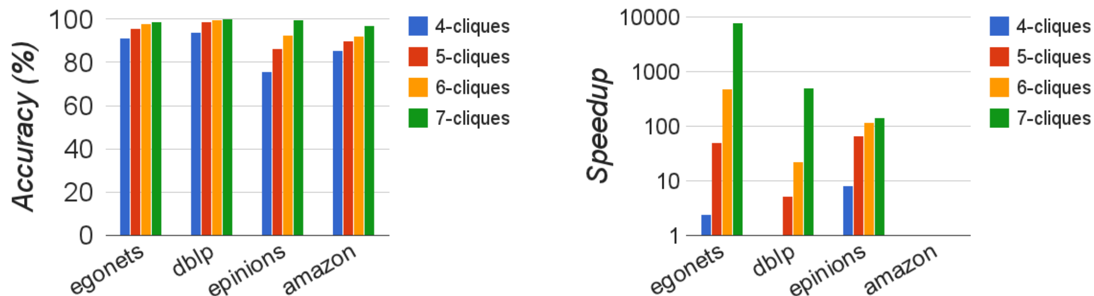

Figure 2.3: (Left) Accuracy of $\mathsf{CES}(r)$ for various graphs. The y-axis is the percentage accuracy, which is the $\mathsf{CES}(r)$ estimate divided by the true answer. (Right) Speedup of CES over a tuned clique enumeration Enum. Bars that are not present represent no speedup, in all cases because both CES and Enum are taking less than a minute.

Our algorithm scales to millions of edges and approximates r-clique counts for r up to 9. It runs orders of magnitude faster than enumeration schemes. For example, for 8-clique counts on a `dblp` graph with one million edges, a tuned enumeration algorithm takes more than a day, while the CES algorithm terminates in minutes.

Summary of Empirical Results We run the CES algorithm on a variety of publicly available social networks from the SNAP database [SNA]. In Figure 2.3 (left), we show the percentage accuracy of $\mathsf{CES}(r)$ on a number of graphs for r up to 7. This shows the CES estimate divided by the true r-clique (or K_r) count. Barring one K_4-count, the accuracy is more than 85%, and is mostly above 90%. The speedup over a fairly well-tuned enumeration is shown in Figure 2.3 (right). In almost all cases shown, CES terminates in minutes and is orders of magnitude faster than enumeration. (For the `amazon` graph, speedup is negligible because both our algorithm and the enumeration algorithm run in under a minute.)

Additionally, for K_8 and K_9 counting, enumeration methods do not terminate in a day, while CES returns estimates mostly within ten minutes. We were able to get K_9-estimates on a `flickr` graph with more than 10^{19} such cliques in less than half an hour. More details are in Section 2.7.3.

2.7.1 Previous Work in Clique Counting

Clique counting is hard both theoretically and empirically. The state-of-the-art theoretical algorithms have an exponential dependence on r [NS85]. There have been a plethora of techniques for triangle counting, including eigenvalue methods [Tso08], graph sparsification [TKMF09], and wedge sampling [SW05, SPK13], but these are all tailored to triangle counting and do not extend to larger values of r. There are also a number of clever triangle enumeration schemes [Coh09, SV11], but enumeration schemes are inevitably shackled to an exponential dependence on r. Empirically, the number of small cliques can range into the billions even for moderately sized graphs.

Sampling is a powerful technique for pattern counting and has been extensively used for triangles [TKMF09, SPK13]. But there are no general sampling methods for finding or counting r-cliques. The obstacles to sampling algorithms are especially tough in sparse graphs (including most interaction networks), and as r grows, the likelihood of successfully sampling an r-clique becomes vanishingly small, necessitating an infeasible number of trials even to find a single clique.

For larger subgraphs, Bordino et al. [BDGL08] count 3- and 4-vertex patterns in a streaming setting. Numerous results in bioinformatics focus on counting small patterns (including cliques) of up to 6 vertices, but they mostly scale to only thousands of edges [PCJ04, HBPS07, GK07, ADH+08]. Rahman et al. [RBH12] use sampling methods for motif counting up to size 6 and have results for networks of around 100,000 edges.

On the flip side, a well-studied problem is that of finding the *maximum* clique. This is a classic NP-hard problem that is also hard to approximate [Has96, KP06]. Recent empirical work by Rossi et al. give an elegant heuristic that runs extremely fast on real data sets [RGGP13]. Recent work by Tsourakakis et al. [TBG+13] and older results of Andersen and Chellapilla [AC09] study the related problem of finding dense subgraphs.

2.7.2 The CES Algorithm

We now present the $\mathsf{CES}(r)$ algorithm for approximately counting r-cliques. An empirical evaluation for $4 \leq r \leq 9$ follows in Section 2.7.3.

We first run the decomposition procedure from Section 2.3 to obtain a tightly-knit family $V_1, \ldots, V_k$. Let G_i be $G|_{V_i}$, or the graph induced on V_i by G.

We then run the following procedure with $S = G_i$ for each $i \in \{1, \ldots, k\}$:

- Enumerate all the triangles, edges, and vertices of S.
- Run w independent trials of the following sampling experiment:
  - Pick $\lfloor r/3 \rfloor$ random triangles from S, and an additional random vertex or edge from S if r is 1 or 2 (mod 3), respectively.
  - Declare success if all the chosen vertices are distinct and form an r-clique in S.

For example, when $r = 5$, an independent trial picks a random triangle and random edge, and is deemed successful if and only if their vertices form a 5-clique.

A simple transformation converts the fraction of successful trials into an unbiased estimator for the number of r-cliques in each G_i, and then we sum over the G_i to get an estimate of the number of r-cliques in G.

More explicitly, let $M_r(S)$ denote the number of distinct choices that the sampling experiment could make for some $S = G_i$. For example, when $r = 5$, $M_r(S)$ is the number of triangles of S times the number of edges of S. Let W_r denote the number of distinct choices that lead to the same r-clique. For example, there are $\binom{5}{3,2} = 10$ ways to choose a disjoint triangle and edge from a

5-clique, so $W_5 = 10$. The estimate of the procedure for G_i is then the fraction of successful trials multiplied by $M_r(G_i)/W_r$.

The number of trials w controls the variance of the estimate. In our experiments, we found $w =$ 100K (100,000) to be sufficient for good accuracy when $r \leq 6$, and $w =$ 10M (10,000,000) to be sufficient for $r \geq 7$. Furthermore, when G_i is sufficiently small, we simply run a clique enumeration algorithm. In our implementation, we enumerate when G_i has fewer than 50 vertices.

Why it Works

We now motivate the algorithm proposed above, and explain intuitively why it should work.

Intuition #1: Enumeration is too expensive. Why not just count the r-cliques of G directly? Known approaches for exact counting effectively enumerate the r-cliques, and these are unusable once the number of r-cliques is overly large. For example, in a Facebook egonet with 88K edges, there are 10^{11} K_7's (more such counts in Table 2.8). An enumerative approach is impractical even for modest-sized social networks and modest values of r. This motivates *approximately* counting the number of r-cliques, and sampling is arguably the most natural way to do this.

Intuition #2: Accurate sampling requires high density. Why not just sample from the original graph G? The number of trials w we need is inversely proportional to the probability of success of each trial, which has a heavy dependence on the density of the graph being sampled. The decomposition procedure puts a lower bound on the density of each G_i, whereas a typical interaction graph will be orders of magnitude more sparse.

Intuition #3: Almost all cliques in interaction networks are confined to dense subgraphs. The decomposition procedure does not explicitly strive to preserve r-cliques for $r \geq 4$, and Theorem 2.16 does not offer any guarantees about them. Intuitively, though, the algorithm's focus on preserving edge and triangle dense substructures is well-suited for preserving larger cliques as well.

2.7.3 Empirical Evaluation

We do an empirical study to answer the following questions.

1. How accurate is CES on real interaction networks?

2. Does CES give significant speedup over enumeration algorithms?

3. Can we empirically verify some of the intuition for why CES should work?

Datasets: We run experiments on a number of social networks of varying sizes, obtained from SNAP [SNA]; high level details are in Table 2.4. We also experiment on an "outlier" graph `youtube`, which has a transitivity of .002 and is therefore outside the regime that the CES algorithm is designed for. The following descriptions are copied or paraphrased from the SNAP website:

	n	m	K_3	transitivity
egonets	4.0 K	0.088 M	1.6 M	0.2647
dblp	320 K	1.0 M	2.2 M	0.1283
epinions	76 K	0.41 M	1.6 M	0.0229
amazon	340 K	0.93 M	0.67 M	0.07925
youtube	1100 K	3.0 M	3.1 M	0.002081
flickr	110 K	2.3 M	107 M	0.1828

Table 2.4: Details of the data sets.

`egonets`: This dataset consists of "circles" (or "friends lists") from Facebook. Facebook data was collected from survey participants using a Facebook app. The dataset includes node features (profiles), circles, and ego networks.

`dblp`: The DBLP computer science bibliography provides a comprehensive list of research papers in computer science. We construct a co-authorship network where two authors are connected if they publish at least one paper together.

`epinions`: This is the who-trusts-who network of the consumer review site Epinions.com. Members of the site can decide whether to trust each other. The trust relationships form a "Web of Trust" which is then combined with review ratings to determine which reviews are shown to the user.

`amazon`: This network is based on the "Customers Who Bought This Item Also Bought" feature of the retail site Amazon.com. If a product i is frequently co-purchased with product j, the graph contains an undirected edge from i to j. The data was collected by crawling Amazon's website.

`youtube`: Youtube is a video-sharing web site that includes a social network. In the Youtube social network, users form friendships with each other and users can create groups which other users can join.

`flickr`: This dataset is built by forming links between images sharing common metadata from the photo sharing site Flickr.com. Edges are formed between images from the same location, images submitted to the same gallery, group, or set, images sharing common tags, images taken by friends, etc.

Implementation details: We run $\mathsf{CES}(r)$ to count r-cliques for $4 \leq r \leq 9$. For `egonets`, `dblp`, and `amazon`, we set w (the sampling parameter) to be 100K samples for $r < 6$, and 1M samples for $6 \leq r < 9$. For `epinions`, we set w to be 100K for $r < 6$, and 10^r samples for $r \geq 6$. For `flickr`, we set w to be 10M samples for $r < 6$, and 100M samples for $6 \leq r < 9$.

To get exact K_r counts, we implement an enumeration scheme Enum based on recursive backtracking. The search for r-cliques begins with "candidates" that are all $(r-1)$-cliques. Each of these is extended to a K_r by searching for a vertex that is connected to all vertices in the clique. We use vertex orderings to ensure that every clique is found exactly once (rather than $r!$ times). All implementations are in C++ and were run on an Ubuntu machine equipped with a 24-core 2.1 GHz AMD Opteron (6172) processor and 96GB of memory. The K_r counts are given in Table 2.8.

The Bottom Line

Accuracy of CES(r) : We show the accuracy of CES(r) in Figure 2.3 for $r \in [4, 7]$ over several graphs, computed as the output of CES(r) divided by the true answer. The accuracy of CES(r) is above 85% for all but one instance, and is above 90% in most cases. Our accuracy increases with r, and for K_7-counting, the accuracy is more than 95% in every graph. We do not show any accuracy results for either `flickr` or larger r because Enum did not terminate in a day.

Speedup of algorithm: The speedup of the CES(r) is shown in Figure 2.3 (right). For most instances, CES(r) is many orders of magnitude faster than the enumeration algorithm. For K_7-counting in `egonets`, Enum takes more than a day, while our algorithm takes only 19 seconds, with only 1% error. Overall, the speedups are on the order of 100x for the graphs with many r-cliques, which makes many previously infeasible clique-counting problems tractable.

Bars that are not present in Figure 2.3 (right) represent no speedup (on `amazon` and K_4-counting on `dblp`). In all these instances, both CES and Enum take less than a minute. We do not have speedup results for `flickr` or $r \geq 8$, because Enum did not terminate in a day.

Scaling to massive outputs: We run our algorithm on a `flickr` graph with millions of edges and hundreds of millions of triangles. The running times are given in Table 2.5. K_r-counting for `flickr` when $r \geq 5$ is infeasible and takes more than a day, while our estimates require less than half and hour.

r	4	5	6	7	8	9
time (minutes)	11	11	13	13	28	29

Table 2.5: CES(r) running times on `flickr` in minutes. Enumeration takes more than a day for all $r \geq 5$.

We also show the times CES required for K_8 and K_9-counting in Table 2.6. (We ignore `amazon`, since it has no 8-cliques.) Other than `epinions` where CES took 3.5 hours for K_9 counting, all other times are at most 30 minutes. Again, enumeration cannot be done in a day on any of these networks.

	`egonets`	`dblp`	`epinions`	`flickr`
K_8	0.3	5	72	28
K_9	2	26	211	29

Table 2.6: CES(r) K_8 and K_9 running times in minutes. Enumeration takes more than a day on all instances.

High transitivity is essential: As stressed earlier, our algorithm is designed to work on graphs with high transitivity. We highlight this by reporting results on `youtube`, a graph with a low transitivity of 0.002. The accuracies are given in Table 2.7. In this case, the accuracies are very poor, most likely because the extractions are destroying most of the cliques.

r	4	5	6	7	8
% Accuracy	58	59	58	5	0.1

Table 2.7: CES(r) accuracies on `youtube`. The Clean and Extract subroutines destroy most of the r-cliques in `youtube`, which has a low transitivity (0.002).

Inferences from Clique Numbers: Large Dense Subgraphs

Looking at K_r counts in Table 2.8, we see that even the tiny `egonets` graph (88K edges) has more than a *trillion* K_9s. Enumeration methods, no matter how well-engineered, are clearly infeasible here.

	`egonets`	`dblp`	`epinions`	`amazon`	`flickr`
n	4.04 E3	3.17 E5	7.59 E4	3.35 E5	1.06 E5
m	8.82 E4	1.05 E6	4.06 E5	9.26 E5	2.32 E6
K_3	1.61 E6	2.22 E6	1.62 E6	6.67 E5	1.07 E8
K_4	3.00 E7	1.67 E7	5.80 E6	2.76 E5	9.58 E9
K_5	5.18 E8	2.63 E8	1.74 E7	6.16 E4	$\sim$ 8.99 E11
K_6	7.83 E9	4.22 E9	4.57 E7	5.80 E3	$\sim$ 8.01 E13
K_7	1.01 E11	6.1 E10	1.04 E8	3.20 E1	$\sim$ 6.21 E15
K_8	$\sim$ 1.11 E12	$\sim$ 7.77 E11	2.02 E8	0	$\sim$ 4.27 E17
K_9	$\sim$ 1.07 E13	$\sim$ 8.81 E12	$\sim$ 3.10 E8	0	$\sim$ 2.50 E19

Table 2.8: Exact K_r-numbers. Numbers with a tilde in front are estimates from our algorithm, where the enumeration algorithm did not terminate in a day.

The pattern of K_r-counts as a function of r suggests properties about the graphs that would be difficult to detect directly. For `egonets`, `dblp`, and `flickr`, the K_r-counts increase by almost an order of magnitude for each increment of r. This strongly suggests the presence of a very large dense structure in these graphs. Even a clique of size 100 can only account for $\binom{100}{9} \approx 1.9\,\text{E}12$ such K_9s, and there are significantly more K_9s in these graphs.

For `epinions`, the clique numbers increase very slightly, and for `amazon`, there are no K_8s. This suggests that the community structure is much weaker in these graphs, and we hypothesize that this is related to their lower transitivities (0.02 and 0.08, respectively). As a side note, the accuracy (error less than 10%) of the CES algorithm for `amazon` is quite surprising, since the K_6- and K_7-counts are quite small.

Why it Works: An Empirical Validation

We discuss experimental results verifying the insights of Section 2.7.2. For convenience, we focus on just the `dblp` graph, though the results are consistent over all the high transitivity graphs.

Cluster density: In Figure 2.9a, we plot the size versus edge densities of the clusters extracted by the calls to Extract. We plot only the clusters with more than 50 vertices. (The densities of

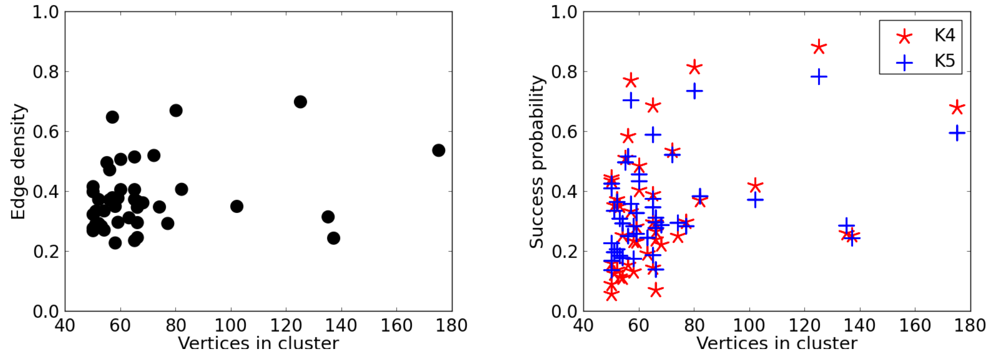

Figure 2.9: Cluster properties of `dblp`. On the left, we plot cluster size vs cluster edge density, for clusters with at least 50 nodes. Note the high density of all clusters. The edge density of the original graph is 2.08×10^{-5}. On the right, we plot the K_4 and K_5 sampling success probabilities by cluster size. Again, note the fairly large success probabilities, versus $\approx 10^{-4}$ and $\approx 10^{-3}$ for K_4 and K_5 sampling in the overall graph.

the smaller clusters are generally even higher.) The edge density of the entire graph is 2.1×10^{-5}, whereas the least dense cluster has density .22, over 10000 times larger. This is consistent with the theoretical underpinnings of the decomposition procedure. These clusters collectively contain 49% of the total edges of the graph.

Sampling success probability: Consider the success probability of a single trial, for both K_4 and K_5. In Figure 2.9b, we plot the size versus these probabilities, again only for clusters of more than 50 vertices. The probabilities are very high, significantly more than 0.1 in almost all clusters. Contrast this with the success probabilities of $\approx 10^{-4}$ and $\approx 10^{-3}$ for K_4 and K_5 sampling on the entire graph.

Full Breakdown of Running Times and Accuracy

The full details of running time and accuracy are presented in Figure 2.10. We also show the breakdown between the running time of the decomposition step (which is independent of r) and that of the sampling procedure.

2.8 Conclusions and Further Directions

This chapter proposes a "model-free" approach to the analysis of social and interaction networks. We restrict attention to graphs that satisfy a combinatorial condition — constant triangle density — in lieu of adopting a particular generative model. The goal of this approach is to develop structural and algorithmic results that apply simultaneously to all reasonable models of interaction networks. Our main result shows that constant triangle density already implies significant graph structure:

		egonets	dblp	epinions	amazon	youtube	flickr
vertices n		4.0K	320K	76K	340K	1100K	110K
edges m		0.088M	1.0M	0.41M	0.93M	3.0M	2.3M
triangles K_3		1.6M	2.2M	1.6M	0.67M	3.1M	107M
transitivity τ		0.2647	0.1283	0.0229	0.07925	0.002081	0.1828
decomposition time		6.28s	22.5s	11.4s	16.2s	70.7s	641s
K_4	brute-force time	17.4s	16.3s	110s	2.98s	519s	641s
	sampling time	0.99s	5.13s	1.87s	1.70s	15.4s	39.3s
	speedup	2.39x	-	8.29x	-	6.03x	40.3x
	%-error	-8.7%	-6.1%	-24.2%	-14.5%	-41.8%	-6.3%
K_5	brute-force time	395s	168s	901s	4.02s	1276s	
	sampling time	1.47s	9.58s	1.80s	1.41s	12.2s	39.4s
	speedup	51.0x	5.24x	68.2x	-	15.4x	
	%-error	-4.4%	-1.3%	-13.7%	-9.9%	-40.7%	
K_6	brute-force time	7548s	2284s	3006s	5.87s	6702s	
	sampling time	9.52s	77.4s	14.3s	5.23s	93.6s	111s
	speedup	478x	22.8x	117x	-	40.8x	
	%-error	-1.9%	-0.5%	-7.5%	-7.8%	-42.0%	
K_7	brute-force time	122995s	50156s	7600s	5.97s	7318s	
	sampling time	12.4s	151s	135s	4.13s	133s	134s
	speedup	7781x	502x	141x	-	44.5x	
	%-error	-1.1%	-0.1%	-0.5%	-3.1%	-95.3%	
K_8	brute-force time			20720s	4.16s	13427s	
	sampling time	13.5s	303s	1385s	4.18s	94.0s	1020s
	speedup			15x	-	81.5x	
	%-error			-3.8%	-	-99.9%	
K_9	brute-force time				4.31s		
	sampling time	105s	1538s	12655s	38.2s	1456s	1101s
	speedup				-		
	%-error				-		

Figure 2.10: Running times and accuracy. The brute-force time is from running Enum on the whole graph (pre-decomposition), and the speedup is (brute-force time)/(decomposition time + sampling time). The blanks are from when Enum did not finish in a day. When it finishes, Enum returns an exact count, and the %-error is the error of CES (= decomposition + sampling) with respect to Enum. The dashes (-) represent either no speedup, or no error for when the number of r-cliques is zero.

every graph that meets this condition is, in a precise sense, well approximated by a disjoint union of clique-like graphs.

Additionally, just as planar separator theorems lead to faster algorithms and better heuristics for planar graphs than for general graphs, we expect our decomposition theorem to be a useful tool in the design of algorithms for triangle-dense graphs. We show an extension of our decomposition procedure that approximates the number of r-cliques in a triangle-dense graph, and verify its efficacy on a variety of publicly-available interaction graphs.

Our work suggests numerous avenues for future research.

1. Can the dependence of the inter-cluster edge and triangle density on the original graph's triangle density be improved?

2. The relative frequencies of four-vertex subgraphs also exhibit special patterns in interaction networks — for example, there are usually very few induced four-cycles [UBK13]. Is there an assumption about four-vertex induced subgraphs, in conjunction with high triangle density, that yields a stronger decomposition theorem?

3. Which other computational problems are easier for triangle-dense graphs than for arbitrary graphs?

4. The approximate clique counting procedure can easily be extended to count other small, dense subgraphs. Does the fast estimation of the frequencies of small substructures allow for broader inference about large substructures or other global properties of a graph?

5. The key parts of both the decomposition procedure and the clique counting extension are naturally parallelizable. How quickly can they be implemented in a MapReduce framework?

6. The decomposition procedure currently runs in time proportional to the number of wedges, which is likely too high for some applications. Is there an approximate version of the procedure that is more efficient?

Finally, our work offers practical advice to those clustering or otherwise operating on large triangle-dense graphs. The cleaning procedure from Section 2.3.2 is easy to implement and can be a natural (and non-obvious) first step for removing "unimportant" or "noisy" edges in a graph. Similarly, the extraction procedure gives good advice for exploring the 2-hop neighborhood around a vertex i — in the notation of Section 2.3.3, rank the vertices j by their θ_j, and then either examine them in order, if one is looking for vertices "similar" to i, or consider the top ℓ vertices for every ℓ up to some threshold, if one is looking for candidate clusters.

2.A Implementation Details

This section describes how to implement the decomposition procedure in the proof of Theorem 2.16 in expected $O(|V|+|E|+w)$ time and $O(|E|)$ space, where w is the number of wedges of the graph.

The implementation includes a few improvements over a naive one. There are three operations that could potentially cause a slowdown: cleaning, finding the maximum degree vertex i, and finding vertices with many triangles incident on the neighborhoods M.

Naively computing all J_e's from scratch for every cleaning procedure could require $\Omega(|E|^2)$ time. To make cleaning fast, we note that $J_{(i,j)}$ only changes when edges incident on i or j get removed. Hence, we maintain a set of "dirtied" vertices after extraction and after each iteration of the cleaning procedure, and only recompute Jaccard similarities for the edges incident on those dirtied vertices. We also keep track of the number of triangles t_e incident on each edge and the degree d_i of each vertex so that computing J_e is an $O(1)$ operation.

To quickly find a maximum-degree vertex every extraction phase, we maintain a lookup from a degree d to the vertices of degree d, and also keep track of the maximum degree $d_{\max}$.

For finding vertices with many triangles incident on the neighborhoods M, we do $O(1)$ work for every triangle with an edge in M. The important point is to enumerate them by computing T_e for each $e \in M$, rather than directly computing the number of triangles from each vertex i, which would take $O(|V|w)$ time.

Sections 2.A.1 and 2.A.2 provide a more detailed description and analysis of the decomposition procedure.

2.A.1 Data structures

We assume the following primitives:

- A *hash table*, or key-value store. With m keys, we assume $O(m)$ space, amortized or expected $O(1)$ time look-up, insertion, and deletion of any key, $O(m)$ enumeration of keys, expected $O(1)$ peek of a "random" key, and $O(1)$ look-up of the total number of keys. We denote hash tables by $\textsc{hash}(\mathcal{K}, k \to f(k))$, where $\mathcal{K}$ is the set of keys, and $k \to f(k)$ is the mapping from keys to values.

- A *set*, or key store. This is simply a degenerate hash table, where every value is 0. Note that the above properties imply $O(m_1 + m_2)$ time intersection of two sets with m_1 and m_2 keys respectively.

- A *multiset*, or key-count store. This is a hash table where the keys are the distinct elements, and the values are counts. We do not store keys with value zero.

Let V' be the set of non-isolated vertices, namely vertices i with $d_i > 0$. We maintain the following data structures throughout the course of the algorithm.

- A hash table $\mathcal{G} = \text{HASH}(V', i \to \text{SET}(N(i)))$ of vertex neighborhoods. This allows $O(N(i))$ lookup of $N(i)$.
- A hash table $\mathcal{D} = \text{HASH}(V', i \to d_i)$ of vertex degrees.
- An array $\mathcal{A}$, where the d'th entry is a pointer to all the vertices of degree d, stored as a set. Note that we implicitly keep track of $|\mathcal{A}[d]|$ via our definition of set.
- An integer $d_{\max} = \max\{d : |\mathcal{A}[d]| > 0\}$.
- A hash table $\mathcal{T} = \text{HASH}(E, e \to t_e)$ that keeps track of the number of triangles incident on e.

Note that we can iterate over all edges in $O(|E|)$ time, via $\mathcal{G}$. We can also iterate over all triangles in $O(w + |E|)$ time, by computing $N(i) \cap N(j)$ for each $(i, j) \in E$.

We define the following operations on the data structures above:

- DELETEEDGE(i, j) removes edge (i, j).
- JACCARD(i, j) returns $J_{(i,j)} = \frac{\mathcal{T}(i,j)}{\mathcal{D}(i)+\mathcal{D}(j)-\mathcal{T}(i,j)}$, and takes $O(1)$ time.
- ISEMPTYGRAPH() returns True if $d_{\max}$ is 0, and takes $O(1)$ time.
- MAXDEGREEVERTEX() returns a vertex from $\mathcal{A}(d_{\max})$, and takes expected $O(1)$ time.

2.A.2 Procedure

We first define the two subroutines below.

CLEAN starts with a "dirty" set of vertices $\mathcal{V}$. It iterates over edges incident on $\mathcal{V}$ until it finds one with $J_{(i,j)} < \epsilon$, after which it deletes (i, j), adds i and j to $\mathcal{V}$, and starts over. If it iterates over all the edges of some $i \in \mathcal{V}$ without finding one with low Jaccard similarity, it removes i from $\mathcal{V}$.

```
1 function CLEAN(𝒱, ϵ)
2     while 𝒱 is non-empty do                    ▷ 𝒱 is the set of dirty vertices
3         for i ∈ 𝒱 do
4             for j ∈ N(i) do
5                 if JACCARD(i, j) < ϵ then
6                     DELETEEDGE(e)
7                     Add j to 𝒱 and go to line 2.
8             Remove i from 𝒱.
```

Recall that θ_j is the number of triangles incident on j whose other two vertices are in $N(i)$. EXTRACT computes θ_j for each j where $\theta_j > 0$, by iterating over T_e for every edge $e \in N(i)$. It then takes the largest d_i such θ_js to form the extracted set S. It removes S from the graph, and dirties the vertices in the neighborhood of S for subsequent cleaning.

```
1 function EXTRACT
2     i = MAXDEGREEVERTEX()
```

```
3      θ = a multiset over vertices                    ▷ θ(j) will count the number of triangles
4                                                        in N(i) ∪ j incident on j.
5      for j_1 ∈ N(i) do
6          for j_2 ∈ N(j_1) ∩ N(i) do                   ▷ (j_1, j_2) iterates over the edges of N(i)
7              for j ∈ N(j_1) ∩ N(j_2) do
8                  Add j to θ.
9      R = the D(i) keys of θ with the highest counts
10     S = R ∪ N(i)
11     V = (⋃_{s∈S} N(s)) \ S                                              ▷ dirtied vertices
12     for e incident on S do
13         DeleteEdge(e)
14     return S, V
```

Finally, we glue the two subroutines together to get the main function below. Partition alternatively Cleans and Extracts until the graph is gone.

```
1  function Partition(G, ε)
2      Construct the data structures G, D, A, d_max, T from G.
3      P = empty list                                           ▷ stores the partition
4      Clean(V′, ε)
5      while not IsEmptyGraph() do
6          S, V = Extract()
7          Append S to P.
8          Clean(V, ε)
9      return (P)
```

We are now ready to prove the main theorem of this section.

Theorem 2.32. *The procedure above runs in expected $O(|V| + |E| + w)$ time and $O(|E|)$ space.*

Proof. The space bound is easy to check, so we focus on the time. We look at the total amount of time spent in DeleteEdge, Clean, Extract, and Partition in order; for each function, we ignore time spent in subroutines that have been previously accounted for. The *total cost* of a function or line of code refers to the time spent there over the course of the entire algorithm.

DeleteEdge: Each edge gets deleted exactly once. It is easy to check that $\mathcal{G}, \mathcal{D}$, and $\mathcal{A}$ each spend $O(|E|)$ time each. The total time spent by $d_{\max}$ is $O(|V|)$; every time $|\mathcal{A}(d_{\max})|$ drops to 0, we do a linear search downwards till we find a non-empty entry of $\mathcal{A}$. The total time spent by $\mathcal{T}$ is at most $\sum_{(i,j)\in E} d_i + d_j = O(|E| + w)$; on deletion of (i, j), $\mathcal{T}(e)$ is decremented for $e = \{(i, k), (j, k) : k \in N(i) \cap N(j)\}$.

Clean: Say a vertex k gets "chosen" every time $i = k$ in lines 4–8. Each vertex $i \in V'$ gets chosen once from line 4 of Partition, and at most once each time d_i changes to a positive value. Since d_i only decreases, each i is chosen at most d_i times. The cost of choosing a vertex i is at most d_i, so the total cost here is at most $\sum_{i\in V} d_i^2 = O(|E| + w)$.

Extract: Over the course of the algorithm each edge ends up as a (j_1, i) pair at most once, so

the total cost of lines 5–6 is at most $\sum_{(j_1,i)\in E} d_{j_1} + d_i = O(|E| + w)$. Similarly, each edge ends up as a (j_1, j_2) pair at most once, and so the total cost of lines 7–8 is also $O(|E| + w)$.

Line 9 can be computed in time proportional to the number of keys in θ. Each vertex i can be in θ at most d_i times, since once $i \in \theta$ it loses at least one edge to lines 12–13. Hence the total cost of line 9 is $O(|E|)$.

Finally, each vertex can be in S at most once, and so the total cost of line 11 is $O(|E|)$.

Partition: The only non-trivial step here is line 2. Initial construction of $\mathcal{G}, \mathcal{D}, \mathcal{A}$ and $d_{\max}$ takes $O(|V| + |E|)$ time, if the graph is originally presented as either as a list of edges or as an adjacency list. Initial construction of $\mathcal{T}$ takes $O(|E| + w)$ time, via iterating over all triangles. □

Chapter 3

Application-Specific Algorithm Selection

Chapter Summary: The best algorithm for a computational problem generally depends on the "relevant inputs," a concept that depends on the application domain and often defies formal articulation. While there is a large literature on empirical approaches to selecting the best algorithm for a given application domain, there has been surprisingly little theoretical analysis of the problem.

This chapter adapts concepts from statistical and online learning theory to reason about application-specific algorithm selection. Our models capture several state-of-the-art empirical and theoretical approaches to the problem, ranging from self-improving algorithms to empirical performance models, and our results identify conditions under which these approaches are guaranteed to perform well. We present one framework that models algorithm selection as a statistical learning problem, and our work here shows that dimension notions from statistical learning theory, historically used to measure the complexity of classes of binary- and real-valued functions, are relevant in a much broader algorithmic context. We also study the online version of the algorithm selection problem, and give possibility and impossibility results for the existence of no-regret learning algorithms.

3.1 Introduction

Rigorously comparing algorithms is hard. The most basic reason for this is that two different algorithms for a computational problem generally have incomparable performance: one algorithm is better on some inputs, but worse on the others. How can a theory advocate one of the algorithms over the other? The simplest and most common solution in the theoretical analysis of algorithms is to summarize the performance of an algorithm using a single number, such as its worst-case performance or its average-case performance with respect to an input distribution. This approach effectively advocates using the algorithm with the best summarizing value (e.g., the smallest worst-case running time).

Solving a problem "in practice" generally means identifying an algorithm that works well for most or all instances of interest. When the "instances of interest" are easy to specify formally in advance

— say, planar graphs — the traditional analysis approaches often give accurate performance predictions and identify useful algorithms. However, instances of interest commonly possess domain-specific features that defy formal articulation. Solving a problem in practice can require selecting an algorithm that is optimized for the specific application domain, even though the special structure of its instances is not well understood. While there is a large literature, spanning numerous communities, on empirical approaches to algorithm selection (e.g. [Fin98, HXHL14, HRG+01, HJY+10, KGM12, LNS09]), there has been surprisingly little theoretical analysis of the problem. One possible explanation is that worst-case analysis, which is the dominant algorithm analysis paradigm in theoretical computer science, is deliberately application-agnostic.

This paper demonstrates that application-specific algorithm selection can be usefully modeled as a learning problem. Our models are straightforward to understand, but also expressive enough to capture several existing approaches in the theoretical computer science and AI communities, ranging from the design and analysis of self-improving algorithms [ACCL06] to the application of empirical performance models [HXHL14].

We present one framework that models algorithm selection as a statistical learning problem in the spirit of [Hau92]. We prove that many useful families of algorithms, including broad classes of greedy and local search heuristics, have small pseudo-dimension and hence low generalization error. Previously, the pseudo-dimension (and the VC dimension, fat shattering dimension, etc.) has been used almost exclusively to quantify the complexity of classes of prediction functions (e.g. [AB99]).[1] Our results demonstrate that this concept is useful and relevant in a much broader algorithmic context. It also offers a novel approach to formalizing the oft-mentioned but rarely-defined "simplicity" of a family of algorithms.

We also study regret-minimization in the online version of the algorithm selection problem. We show that the "non-Lipschitz" behavior of natural algorithm classes precludes learning algorithms that have no regret in the worst case, and prove positive results under smoothed analysis-type assumptions.

Paper Organization Section 3.2 outlines a number of concrete problems that motivate the present work, ranging from greedy heuristics to SAT solvers, and from self-improving algorithms to parameter tuning. The reader interested solely in the technical development can skip this section with little loss. Section 3.3 models the task of determining the best application-specific algorithm as a PAC learning problem, and brings the machinery of statistical learning theory to bear on a wide class of problems, including greedy heuristic selection, sorting, and gradient descent step size selection. A time-limited reader can glean the gist of our contributions from Sections 3.3.1–3.3.3. Section 3.4 considers the problem of learning an application-specific algorithm online, with the goal of minimizing regret. Sections 3.4.2 and 3.4.3 present negative and positive results for worst-case and smoothed instances, respectively. Section 3.5 concludes with a number of open research directions.

[1] A few exceptions: [SBD06] use the pseudo-dimension to study the problem of learning a good kernel for use in a support vector machine, [Lon01] parameterizes the performance of the randomized rounding of packing and covering linear programs by the pseudo-dimension of a set derived from the constraint matrix, and [MM14, MR15] use dimension notions from learning theory to bound the sample complexity of learning approximately revenue-maximizing truthful auctions.

3.2 Motivating Scenarios

Our learning framework sheds light on several well-known approaches, spanning disparate application domains, to the problem of learning a good algorithm from data. To motivate and provide interpretations of our results, we describe several of these in detail.

3.2.1 Example #1: Greedy Heuristic Selection

One of the most common and also most challenging motivations for algorithm selection is presented by computationally difficult optimization problems. When the available computing resources are inadequate to solve such a problem exactly, heuristic algorithms must be used. For most hard problems, our understanding of when different heuristics work well remains primitive. For concreteness, we describe one current and high-stakes example of this issue, which also aligns well with our model and results in Section 3.3.3. The computing and operations research literature has many similar examples.

The FCC is currently (in 2016) running a novel double auction to buy back licenses for spectrum from certain television broadcasters and resell them to telecommunication companies for wireless broadband use. The auction is expected to generate over $20 billion dollars for the US government [CBO14]. The "reverse" (i.e., buyback) phase of the auction must determine which stations to buy out (and what to pay them). The auction is tasked with buying out sufficiently many stations so that the remaining stations (who keep their licenses) can be "repacked" into a small number of channels, leaving a target number of channels free to be repurposed for wireless broadband. To first order, the feasible repackings are determined by interference constraints between stations. Computing a repacking therefore resembles familiar hard combinatorial problems like the independent set and graph coloring problems. The reverse auction uses a greedy heuristic to compute the order in which stations are removed from the reverse auction (removal means the station keeps its license) [MS14]. The chosen heuristic favors stations with high value, and discriminates against stations that interfere with a large number of other stations.[2] There are many ways of combining these two criteria, and no obvious reason to favor one specific implementation over another. The specific implementation in the FCC auction has been justified through trial-and-error experiments using synthetic instances that are thought to be representative [MS14]. One interpretation of our results in Section 3.3.3 is as a post hoc justification of this exhaustive approach for sufficiently simple classes of algorithms, including the greedy heuristics considered for this FCC auction.

3.2.2 Example #2: Self-Improving Algorithms

The area of *self-improving algorithms* was initiated by [ACCL06], who considered sorting and clustering problems. Subsequent work [CS08, CMS10, CMS12] studied several problems in low-dimensional geometry, including the maxima and convex hull problems. For a given problem, the

[2] Analogously, greedy heuristics for the maximum-weight independent set problem favor vertices with higher weights and with lower degrees [STY03]. Greedy heuristics for welfare maximization in combinatorial auctions prefer bidders with higher values and smaller demanded bundles [LOS02].

goal is to design an algorithm that, given a sequence of i.i.d. samples from an unknown distribution over instances, converges to the optimal algorithm for that distribution. In addition, the algorithm should use only a small amount of auxiliary space. For example, for sorting independently distributed array entries, the algorithm in [ACCL06] solves each instance (on n numbers) in $O(n \log n)$ time, uses space $O(n^{1+c})$ (where $c > 0$ is an arbitrarily small constant), and after a polynomial number of samples has expected running time within a constant factor of that of an information-theoretically optimal algorithm for the unknown input distribution. Section 3.3.4 reinterprets self-improving algorithms via our general framework.

3.2.3 Example #3: Parameter Tuning in Optimization and Machine Learning

Many "algorithms" used in practice are really meta-algorithms, with a large number of free parameters that need to be instantiated by the user. For instance, implementing even in the most basic version of gradient descent requires choosing a step size and error tolerance. For a more extreme version, CPLEX, a widely-used commercial linear and integer programming solver, comes with a 221-page parameter reference manual describing 135 parameters [XHHL11].

An analogous problem in machine learning is "hyperparameter optimization," where the goal is to tune the parameters of a learning algorithm so that it learns (from training data) a model with high accuracy on test data, and in particular a model that does not overfit the training data. A simple example is regularized regression, such as ridge regression, where a single parameter governs the trade-off between the accuracy of the learned model on training data and its "complexity." More sophisticated learning algorithms can have many more parameters.

Figuring out the "right" parameter values is notoriously challenging in practice. The CPLEX manual simply advises that "you may need to experiment with them." In machine learning, parameters are often set by discretizing and then applying brute-force search (a.k.a. "grid search"), perhaps with random subsampling ("random search") [BB12]. When this is computationally infeasible, variants of gradient descent are often used to explore the parameter space, with no guarantee of converging to a global optimum.

The results in Section 3.3.6 can be interpreted as a sample complexity analysis of grid search for the problem of choosing the step size in gradient descent to minimize the expected number of iterations needed for convergence. We view this as a first step toward reasoning more generally about the problem of learning good parameters for machine learning algorithms.

3.2.4 Example #4: Empirical Performance Models for SAT Algorithms

The examples above already motivate selecting an algorithm for a problem based on characteristics of the application domain. A more ambitious and refined approach is to select an algorithm on a *per-instance* (instead of a per-domain) basis. While it's impossible to memorize the best algorithm for every possible instance, one might hope to use coarse *features* of a problem instance as a guide to which algorithm is likely to work well.

For example, [XHHL08] applied this idea to the satisfiability (SAT) problem. Their algorithm portfolio consisted of seven state-of-the-art SAT solvers with incomparable and widely varying running times across different instances. The authors identified a number of instance features, ranging from simple features like input size and clause/variable ratio, to complex features like Knuth's estimate of the search tree size [Knu75] and the rate of progress of local search probes.[3] The next step involved building an "empirical performance model" (EPM) for each of the seven algorithms in the portfolio — a mapping from instance feature vectors to running time predictions. They then computed their EPMs using labeled training data and a suitable regression model. With the EPMs in hand, it is clear how to perform per-instance algorithm selection: given an instance, compute its features, use the EPMs to predict the running time of each algorithm in the portfolio, and run the algorithm with the smallest predicted running time. Using these ideas (and several optimizations), their "SATzilla" algorithm won numerous medals at the 2007 SAT Competition.[4] Section 3.3.5 outlines how to extend our PAC learning framework to reason about EPMs and feature-based algorithm selection.

3.3 PAC Learning an Application-Specific Algorithm

This section casts the problem of selecting the best algorithm for a poorly understood application domain as one of learning the optimal algorithm with respect to an unknown instance distribution. Section 3.3.1 formally defines the basic model, Section 3.3.2 reviews relevant preliminaries from statistical learning theory, Section 3.3.3 bounds the pseudo-dimension of many classes of greedy and local search heuristics, Section 3.3.4 re-interprets the theory of self-improving algorithms via our framework, Section 3.3.5 extends the basic model to capture empirical performance models and feature-based algorithm selection, and Section 3.3.6 studies step size selection in gradient descent.

3.3.1 The Basic Model

Our basic model consists of the following ingredients.

1. A fixed computational or optimization problem Π. For example, Π could be computing a maximum-weight independent set of a graph (Section 3.2.1), or sorting n elements (Section 3.2.2).
2. An unknown distribution $\mathcal{D}$ over instances $x \in \Pi$.
3. A set $\mathcal{A}$ of algorithms for Π; see Sections 3.3.3 and 3.3.4 for concrete examples.
4. A performance measure $\textsc{cost} : \mathcal{A} \times \Pi \to [0, H]$ indicating the performance of a given algorithm on a given instance. Two common choices for $\textsc{cost}$ are the running time of an algorithm, and, for optimization problems, the objective function value of the solution produced by an algorithm.

[3] It is important, of course, that computing the features of an instance is an easier problem than solving it.
[4] See [XHHL12] for details on the latest generation of their solver.

The "application-specific information" is encoded by the unknown input distribution $\mathcal{D}$, and the corresponding "application-specific optimal algorithm" $A_{\mathcal{D}}$ is the algorithm that minimizes or maximizes (as appropriate) $\mathbf{E}_{x \in \mathcal{D}}[\text{COST}(A, x)]$ over $A \in \mathcal{A}$. The *error* of an algorithm $A \in \mathcal{A}$ for a distribution $\mathcal{D}$ is

$$\left|\mathbf{E}_{x \sim \mathcal{D}}[\text{COST}(A, x)] - \mathbf{E}_{x \sim \mathcal{D}}[\text{COST}(A_{\mathcal{D}}, x)]\right|.$$

In our basic model, the goal is:

Learn the application-specific optimal algorithm from data (i.e., samples from $\mathcal{D}$).

More precisely, the learning algorithm is given m i.i.d. samples $x_1, \ldots, x_m \in \Pi$ from $\mathcal{D}$, and (perhaps implicitly) the corresponding performance $\text{COST}(A, x_i)$ of each algorithm $A \in \mathcal{A}$ on each input x_i. The learning algorithm uses this information to suggest an algorithm $\hat{A} \in \mathcal{A}$ to use on future inputs drawn from $\mathcal{D}$. We seek learning algorithms that almost always output an algorithm of $\mathcal{A}$ that performs almost as well as the optimal algorithm in $\mathcal{A}$ for $\mathcal{D}$.

Definition 3.1. *A learning algorithm L (ϵ, δ)-learns the optimal algorithm in $\mathcal{A}$ from m samples if, for every distribution $\mathcal{D}$ over Π, with probability at least $1 - \delta$ over m samples $x_1, \ldots, x_m \sim \mathcal{D}$, L outputs an algorithm $\hat{A} \in \mathcal{A}$ with error at most ϵ.*

3.3.2 Pseudo-Dimension and Uniform Convergence

PAC learning an optimal algorithm, in the sense of Definition 3.1, reduces to bounding the "complexity" of the class $\mathcal{A}$ of algorithms. We next review the relevant definitions from statistical learning theory.

Let $\mathcal{H}$ denote a set of real-valued functions defined on the set X. A finite subset $S = \{x_1, \ldots, x_m\}$ of X is *(pseudo-)shattered* by $\mathcal{H}$ if there exist real-valued *witnesses* $r_1, \ldots, r_m$ such that, for each of the 2^m subsets T of S, there exists a function $h \in \mathcal{H}$ such that $h(x_i) > r_i$ if and only if $i \in T$ (for $i = 1, 2, \ldots, m$). The *pseudo-dimension* of $\mathcal{H}$ is the cardinality of the largest subset shattered by $\mathcal{H}$ (or $+\infty$, if arbitrarily large finite subsets are shattered by $\mathcal{H}$). The pseudo-dimension is a natural extension of the VC dimension from binary-valued to real-valued functions.[5]

To bound the sample complexity of accurately estimating the expectation of all functions in $\mathcal{H}$, with respect to an arbitrary probability distribution $\mathcal{D}$ on X, it is enough to bound the pseudo-dimension of $\mathcal{H}$.

Theorem 3.2 (Uniform Convergence (e.g. [AB99]))**.** *Let $\mathcal{H}$ be a class of functions with domain X and range in $[0, H]$, and suppose $\mathcal{H}$ has pseudo-dimension $d_{\mathcal{H}}$. For every distribution $\mathcal{D}$ over X, every $\epsilon > 0$, and every $\delta \in (0, 1]$, if*

$$m \geq c \left(\frac{H}{\epsilon}\right)^2 \left(d_{\mathcal{H}} + \ln\left(\frac{1}{\delta}\right)\right) \tag{3.1}$$

[5]The *fat shattering dimension* is another common extension of the VC dimension to real-valued functions. It is a weaker condition, in that the fat shattering dimension of $\mathcal{H}$ is always at most the pseudo-dimension of $\mathcal{H}$, that is still sufficient for sample complexity bounds. Most of our arguments give the same upper bounds on pseudo-dimension and fat shattering dimension, so we present the stronger statements.

for a suitable constant c (independent of all other parameters), then with probability at least $1-\delta$ *over* m *samples* $x_1, \ldots, x_m \sim \mathcal{D}$*,*

$$\left| \left(\frac{1}{m} \sum_{i=1}^{m} h(x_i) \right) - \mathbf{E}_{x \sim \mathcal{D}}[h(x)] \right| < \epsilon$$

for every $h \in \mathcal{H}$*.*

We can identify each algorithm $A \in \mathcal{A}$ with the real-valued function $x \mapsto \text{COST}(A, x)$. Regarding the class $\mathcal{A}$ of algorithms as a set of real-valued functions defined on Π, we can discuss its pseudo-dimension, as defined above. We need one more definition before we can apply our machinery to learn algorithms from $\mathcal{A}$.

Definition 3.3 (Empirical Risk Minimization (ERM))**.** *Fix an optimization problem* Π*, a performance measure* COST*, and a set of algorithms* $\mathcal{A}$*. An algorithm* L *is an* ERM algorithm *if, given any finite subset* S *of* Π*,* L *returns an (arbitrary) algorithm from* $\mathcal{A}$ *with the best average performance on* S*.*

For example, for any Π, COST, and finite $\mathcal{A}$, there is the trivial ERM algorithm that simply computes the average performance of each algorithm on S by brute force, and returns the best one. The next corollary follows easily from Definition 3.1, Theorem 3.2, and Definition 3.3.

Corollary 3.4. *Fix parameters* $\epsilon > 0$*,* $\delta \in (0, 1]$*, a set of problem instances* Π*, and a performance measure* COST*. Let* $\mathcal{A}$ *be a set of algorithms that has pseudo-dimension* d *with respect to* Π *and* COST*. Then any ERM algorithm* $(2\epsilon, \delta)$*-learns the optimal algorithm in* $\mathcal{A}$ *from* m *samples, where* m *is defined as in* (3.1)*.*

Corollary 3.4 is only interesting if interesting classes of algorithms $\mathcal{A}$ have small pseudo-dimension. In the simple case where $\mathcal{A}$ is finite, as in our example of an algorithm portfolio for SAT (Section 3.2.4), the pseudo-dimension of $\mathcal{A}$ is trivially at most $\log_2 |\mathcal{A}|$. The following sections demonstrate the much less obvious fact that natural infinite classes of algorithms also have small pseudo-dimension.

Remark 3.5 (Computational Efficiency)**.** *The present work focuses on the sample complexity rather than the computational aspects of learning, so outside of a few remarks we won't say much about the existence or efficiency of ERM in our examples. A priori, an infinite class of algorithms may not admit any ERM algorithm at all, though all of the examples in this chapter do have ERM algorithms under mild assumptions.*

3.3.3 Application: Greedy Heuristics and Extensions

The goal of this section is to bound the pseudo-dimension of many classes of greedy heuristics including, as a special case, the family of heuristics relevant for the FCC double auction described in Section 3.2.1. It will be evident that analogous computations are possible for many other classes of heuristics, and we provide several extensions in Section 3.3.3 to illustrate this point. Throughout this section, the performance measure COST is the objective function value of the solution produced by a heuristic on an instance, where we assume without loss of generality a maximization objective.

Definitions and Examples

Our general definitions are motivated by greedy heuristics for (NP-hard) problems like the following; the reader will have no difficulty coming up with additional natural examples.

1. *Knapsack.* The input is n items with values $v_1, \ldots, v_n$, sizes $s_1, \ldots, s_n$, and a knapsack capacity C. The goal is to compute a subset $S \subseteq \{1, 2, \ldots, n\}$ with maximum total value $\sum_{i \in S} v_i$, subject to having total size $\sum_{i \in S} s_i$ at most C. Two natural greedy heuristics are to greedily pack items (subject to feasibility) in order of nonincreasing value v_i, or in order of nonincreasing density v_i/s_i (or to take the better of the two, see Section 3.3.3).

2. *Maximum-Weight Independent Set (MWIS).* The input is an undirected graph $G = (V, E)$ and a non-negative weight w_v for each vertex $v \in V$. The goal is to compute the independent set — a subset of mutually non-adjacent vertices — with maximum total weight. Two natural greedy heuristics are to greedily choose vertices (subject to feasibility) in order of nonincreasing weight w_v, or nonincreasing density $w_v/(1 + \deg(v))$. (The intuition for the denominator is that choosing v "uses up" $1+\deg(v)$ vertices — v and all of its neighbors.) The latter heuristic also has a (superior) adaptive variant, where the degree $\deg(v)$ is computed in the subgraph induced by the vertices not yet blocked from consideration, rather than in the original graph.[6]

3. *Machine Scheduling.* This is a family of optimization problems, where n jobs with various attributes (processing time, weight, deadline, etc.) need to be assigned to m machines, perhaps subject to some constraints (precedence constraints, deadlines, etc.), to optimize some objective (makespan, weighted sum of completion times, number of late jobs, etc.). A typical greedy heuristic for such a problem considers jobs in some order according to a score derived from the job parameters (e.g., weight divided by processing time), subject to feasibility, and always assigns the current job to the machine that currently has the lightest load (again, subject to feasibility).

In general, we consider *object assignment problems*, where the input is a set of n objects with various attributes, and the feasible solutions consist of assignments of the objects to a finite set R, subject to feasibility constraints. The attributes of an object are represented as an element ξ of an abstract set. For example, in the Knapsack problem ξ encodes the value and size of an object; in the MWIS problem, ξ encodes the weight and (original or residual) degree of a vertex. In the Knapsack and MWIS problems, $R = \{0, 1\}$, indicating whether or not a given object is selected. In machine scheduling problems, R could be $\{1, 2, \ldots, m\}$, indicating the machine to which a job is assigned, or a richer set that also keeps track of the job ordering on each machine.

By a *greedy heuristic*, we mean algorithms of the following form (cf., the "priority algorithms" of [BNR03]):

1. While there remain unassigned objects:

   (a) Use a *scoring rule* σ (see below) to compute a score $\sigma(\xi_i)$ for each unassigned object i, as a function of its current attributes ξ_i.

[6] An equivalent description is: whenever a vertex v is added to the independent set, delete v and its neighbors from the graph, and recurse on the remaining graph.

(b) For the unassigned object i with the highest score, use an *assignment rule* to assign i a value from R and, if necessary, update the attributes of the other unassigned objects.[7] For concreteness, assume that ties are always resolved lexicographically.

A *scoring rule* assigns a real number to an object as a function of its attributes. Assignment rules that do not modify objects' attributes yield non-adaptive greedy heuristics, which use only the original attributes of each object (like v_i or v_i/s_i in the Knapsack problem, for instance). In this case, objects' scores can be computed in advance of the main loop of the greedy heuristic. Assignment rules that modify object attributes yield adaptive greedy heuristics, such as the adaptive MWIS heuristic described above.

In a *single-parameter* family of scoring rules, there is a scoring rule of the form $\sigma(\rho, \xi)$ for each parameter value ρ in some interval $I \subseteq \mathbb{R}$. Moreover, σ is assumed to be continuous in ρ for each fixed value of ξ. Natural examples include Knapsack scoring rules of the form v_i/s_i^ρ and MWIS scoring rules of the form $w_v/(1+\deg(v))^\rho$ for $\rho \in [0,1]$ or $\rho \in [0, \infty)$. A single-parameter family of scoring rules is *κ-crossing* if, for each distinct pair of attributes ξ, ξ', there are at most κ values of ρ for which $\sigma(\rho, \xi) = \sigma(\rho, \xi')$. For example, all of the scoring rules mentioned above are 1-crossing rules.

For an example assignment rule, in the Knapsack and MWIS problems, the rule simply assigns i to "1" if it is feasible to do so, and to "0" otherwise. A typical machine scheduling assignment rule assigns the current job to the machine with the lightest load. In the adaptive greedy heuristic for the MWIS problem, whenever the assignment rule assigns "1" to a vertex v, it updates the residual degrees of other unassigned vertices (two hops away) accordingly.

We call an assignment rule *β-bounded* if every object i is guaranteed to take on at most β distinct attribute values. For example, an assignment rule that never modifies an object's attributes is 1-bounded. The assignment rule in the adaptive MWIS algorithm is n-bounded, since it only modifies the degree of a vertex (which lies in $\{0, 1, 2 \ldots, n-1\}$).

Coupling a single-parameter family of κ-crossing scoring rules with a fixed β-bounded assignment rule yields a *(κ, β)-single-parameter family of greedy heuristics.* All of our running examples of greedy heuristics are $(1,1)$-single-parameter families, except for the adaptive MWIS heuristic, which is a $(1, n)$-single-parameter family.

Upper Bound on Pseudo-Dimension

We next show that every (κ, β)-single-parameter family of greedy heuristics has small pseudo-dimension. This result applies to all of the concrete examples mentioned above, and it is easy to come up with other examples (for the problems already discussed, and for additional problems).

Theorem 3.6 (Pseudo-Dimension of Greedy Algorithms)**.** *If $\mathcal{A}$ denotes a (κ, β)-single-parameter family of greedy heuristics for an object assignment problem with n objects, then the pseudo-dimension of $\mathcal{A}$ is $O(\log(\kappa\beta n))$.*

[7] We assume that there is always as least one choice of assignment that respects the feasibility constraints; this holds for all of our motivating examples.

In particular, all of our running examples are classes of heuristics with pseudo-dimension $O(\log n)$.

Proof. Recall from the definitions (Section 3.3.2) that we need to upper bound the size of every set that is shatterable using the greedy heuristics in $\mathcal{A}$. For us, a set is a fixed set of s inputs (each with n objects) $S = x_1, \ldots, x_s$. For a potential witness $r_1, \ldots, r_s \in \mathbb{R}$, every algorithm $A \in \mathcal{A}$ induces a binary labeling of each sample x_i, according to whether $\text{COST}(A, x_i)$ is strictly more than or at most r_i. We proceed to bound from above the number of distinct binary labellings of S induced by the algorithms of $\mathcal{A}$, for any potential witness.

Consider ranging over algorithms $A \in \mathcal{A}$ — equivalently, over parameter values $\rho \in I$. The trajectory of a greedy heuristic $A \in \mathcal{A}$ is uniquely determined by the outcome of the comparisons between the current scores of the unassigned objects in each iteration of the algorithm. Since the family uses a κ-crossing scoring rule, for every pair i, j of distinct objects and possible attributes ξ_i, ξ_j, there are at most κ values of ρ for which there is a tie between the score of i (with attributes ξ_i) and that of j (with attributes ξ_j). Since σ is continuous in ρ, the relative order of the score of i (with ξ_i) and j (with ξ_j) remains the same in the open interval between two successive values of ρ at which their scores are tied. The upshot is that we can partition I into at most $\kappa + 1$ intervals such that the outcome of the comparison between i (with attributes ξ_i) and j (with attributes ξ_j) is constant on each interval.[8]

Next, the s instances of S contain a total of sn objects. Each of these objects has some initial attributes. Because the assignment rule is β-bounded, there are at most $sn\beta$ object-attribute pairs (i, ξ_i) that could possibly arise in the execution of any algorithm from $\mathcal{A}$ on any instance of S. This implies that, ranging across all algorithms of $\mathcal{A}$ on all inputs in S, comparisons are only ever made between at most $(sn\beta)^2$ pairs of object-attribute pairs (i.e., between an object i with current attributes ξ_i and an object j with current attributes ξ_j). We call these the *relevant comparisons*.

For each relevant comparison, we can partition I into at most $\kappa + 1$ subintervals such that the comparison outcome is constant (in ρ) in each subinterval. Intersecting the partitions of all of the at most $(sn\beta)^2$ relevant comparisons splits I into at most $(sn\beta)^2\kappa + 1$ subintervals such that *every* relevant comparison is constant in each subinterval. That is, all of the algorithms of $\mathcal{A}$ that correspond to the parameter values ρ in such a subinterval execute identically on every input in S. The number of binary labellings of S induced by algorithms of $\mathcal{A}$ is trivially at most the number of such subintervals. Our upper bound $(sn\beta)^2\kappa + 1$ on the number of subintervals exceeds 2^s, the requisite number of labellings to shatter S, only if $s = O(\log(\kappa\beta n))$. □

Theorem 3.6 and Corollary 3.4 imply that, if κ and β are bounded above by a polynomial in n, then an ERM algorithm (ϵ, δ)-learns the optimal algorithm in $\mathcal{A}$ from only $m = \tilde{O}(\frac{H^2}{\epsilon^2})$ samples,[9] where H is the largest objective function value of a feasible solution output by an algorithm of $\mathcal{A}$ on an instance of Π.[10]

[8] This argument assumes that $\xi_i \neq \xi_j$. If $\xi_i = \xi_j$, then because we break ties between equal scores lexicographically, the outcome of the comparison between $\sigma(\xi_i)$ and $\sigma(\xi_j)$ is in fact constant on the entire interval I of parameter values.

[9] The notation $\tilde{O}(\cdot)$ suppresses logarithmic factors.

[10] Alternatively, the dependence of m on H can be removed if learning error ϵH (rather than ϵ) can be tolerated — for example, if the optimal objective function value is expected to be proportional to H anyways.

We note that Theorem 3.6 gives a quantifiable sense in which natural greedy algorithms are indeed "simple algorithms." Not all classes of algorithms have such a small pseudo-dimension; see also the next section for further discussion.[11]

Remark 3.7 (Non-Lipschitzness). *We noted in Section 3.3.2 that the pseudo-dimension of a finite set $\mathcal{A}$ is always at most $\log_2 |\mathcal{A}|$. This suggests a simple discretization approach to learning the best algorithm from $\mathcal{A}$: take a finite "ϵ-net" of $\mathcal{A}$ and learn the best algorithm in the finite net. (Indeed, Section 3.3.6 uses precisely this approach.) The issue is that without some kind of Lipschitz condition — stating that "nearby" algorithms in $\mathcal{A}$ have approximately the same performance on all instances — there's no reason to believe that the best algorithm in the net is almost as good as the best algorithm from all of $\mathcal{A}$. Two different greedy heuristics — two MWIS greedy algorithms with arbitrarily close ρ-values, say — can have completely different executions on an instance. This lack of a Lipschitz property explains why we take care in Theorem 3.6 to bound the pseudo-dimension of the full infinite set of greedy heuristics.*[12]

Computational Considerations

The proof of Theorem 3.6 also demonstrates the presence of an efficient ERM algorithm: the $O((sn\beta)^2)$ relevant comparisons are easy to identify, the corresponding subintervals induced by each are easy to compute (under mild assumptions on the scoring rule), and brute-force search can be used to pick the best of the resulting $O((sn\beta)^2\kappa)$ algorithms (an arbitrary one from each subinterval). This algorithm runs in polynomial time as long as β and κ are polynomial in n, and every algorithm of $\mathcal{A}$ runs in polynomial time.

For example, for the family of Knapsack scoring rules described above, implementing this ERM algorithm reduces to comparing the outputs of $O(n^2m)$ different greedy heuristics (on each of the m sampled inputs), with $m = O(\log n)$. For the adaptive MWIS heuristics, where $\beta = n$, it is enough to compare the sample performance of $O(n^4m)$ different greedy algorithms, with $m = O(\log n)$.

Extensions: Multiple Algorithms, Multiple Parameters, and Local Search

Theorem 3.6 is robust and its proof is easily modified to accommodate various extensions. For a first example, consider algorithms than run q different members of a single-parameter greedy heuristic family and return the best of the q feasible solutions obtained.[13] Extending the proof of Theorem 3.6 yields a pseudo-dimension bound of $O(q \log(\kappa\beta n))$ for the class of all such algorithms.

For a second example, consider families of greedy heuristics parameterized by d real-valued parameters $\rho_1, \ldots, \rho_d$. Here, an analog of Theorem 3.6 holds with the crossing number κ replaced by a

[11] When the performance measure COST is solution quality, as in this section, one cannot identify "simplicity" with "low pseudo-dimension" without caveats: strictly speaking, the set $\mathcal{A}$ containing only the optimal algorithm for the problem has pseudo-dimension 1. When the problem Π is NP-hard and $\mathcal{A}$ consists only of polynomial-time algorithms (and assuming $P \neq NP$), the pseudo-dimension is a potentially relevant complexity measure for the heuristics in $\mathcal{A}$.

[12] The ϵ-net approach has the potential to work for greedy algorithms that choose the next object using a softmax-type rule, rather than deterministically as the unassigned object with the highest score.

[13] For example, the classical $\frac{1}{2}$-approximation for Knapsack has this form (with $q = 2$).

more complicated parameter — essentially, the number of connected components of the co-zero set of the difference of two scoring functions (with ξ, ξ' fixed and variables $\rho_1, \ldots, \rho_d$). This number can often be bounded (by a function exponential in d) in natural cases, for example using Bézout's theorem (see e.g. [Gat14]).

For a final extension, we sketch how to adapt the definitions and results of this section from greedy to local search heuristics. The input is again an object assignment problem (see Section 3.3.3), along with an initial feasible solution (i.e., an assignment of objects to R, subject to feasibility constraints). By a *k-swap local search heuristic*, we mean algorithms of the following form:

1. Start with arbitrary feasible solution.
2. While the current solution is not locally optimal:
   (a) Use a *scoring rule* σ to compute a score $\sigma(\{\xi_i : i \in K\})$ for each set of objects K of size k, where ξ_i is the current attribute of object i.
   (b) For the set K with the highest score, use an *assignment rule* to re-assign each $i \in K$ to a value from R. If necessary, update the attributes of the appropriate objects. (Again, assume that ties are resolved lexicographically.)

We assume that the assignment rule maintains feasibility, so that we have a feasible assignment at the end of each execution of the loop. We also assume that the scoring and assignment rules ensure that the algorithm terminates, e.g. via the existence of a global objective function that decreases at every iteration (or by incorporating timeouts).

A canonical example of a k-swap local search heuristic is the k-OPT heuristic for the traveling salesman problem (TSP)[14] (see e.g. [JM97]). We can view TSP as an object assignment problem, where the objects are edges and $R = \{0, 1\}$; the feasibility constraint is that the edges assigned to 1 should form a tour. Recall that a local move in k-OPT consists of swapping out k edges from the current tour and swapping in k edges to obtain a new tour. (So in our terminology, k-OPT is a $2k$-swap local search heuristic.) Another well-known example is the local search algorithms for the p-median problem studied in [AGK+04], which are parameterized by the number of medians that can be removed and added in each local move. Analogous local search algorithms make sense for the MWIS problem as well.

Scoring and assignment rules are now defined on subsets of k objects, rather than individual objects. A single-parameter family of scoring rules is now called *κ-crossing* if, for every subset K of at most k objects and each distinct pair of attribute sets ξ_K and ξ'_K, there are at most κ values of ρ for which $\sigma(\rho, \xi_K) = \sigma(\rho, \xi'_K)$. An assignment rule is now *β-bounded* if for every subset K of at most k objects, ranging over all possible trajectories of the local search heuristic, the attribute set of K takes on at most β distinct values. For example, in MWIS, suppose we allow two vertices u, v to be removed and two vertices y, z to be added in a single local move, and we use the single-parameter scoring rule family

$$\sigma_\rho(u, v, y, z) = \frac{w_u}{(1 + \deg(u))^\rho} + \frac{w_v}{(1 + \deg(v))^\rho} - \frac{w_y}{(1 + \deg(y))^\rho} - \frac{w_z}{(1 + \deg(z))^\rho}.$$

[14] Given a complete undirected graph with a cost c_{uv} for each edge (u, v), compute a tour (visiting each vertex exactly once) that minimizes the sum of the edge costs.

Here $\deg(v)$ could refer to the degree of vertex v in original graph, to the number of neighbors of v that do not have any neighbors other than v in the current independent set, etc. In any case, since a generalized Dirichlet polynomial with t terms has at most $t-1$ zeroes (see e.g. [Jam06, Corollary 3.2]), this is a 3-crossing family. The natural assignment rule is n^4-bounded.[15]

By replacing the number n of objects by the number $O(n^k)$ of subsets of at most k objects in the proof of Theorem 3.6, we obtain the following.

Theorem 3.8 (Pseudo-Dimension of Local Search Algorithms)**.** *If $\mathcal{A}$ denotes a (κ, β)-single-parameter family of k-swap local search heuristics for an object assignment problem with n objects, then the pseudo-dimension of $\mathcal{A}$ is $O(k \log(\kappa\beta n))$.*

3.3.4 Application: Self-Improving Algorithms Revisited

We next give a new interpretation of the self-improving sorting algorithm of [ACCL06]. Namely, we show that the main result in [ACCL06] effectively identifies a set of sorting algorithms that simultaneously has low representation error (for independently distributed array elements) and small pseudo-dimension (and hence low generalization error). Other constructions of self-improving algorithms [ACCL06, CS08, CMS10, CMS12] can be likewise reinterpreted. In contrast to Section 3.3.3, here our performance measure COST is related to the running time of an algorithm A on an input x, which we want to minimize, rather than the objective function value of the output, which we wanted to maximize.

Consider the problem of sorting n real numbers in the comparison model. By a *bucket-based sorting algorithm*, we mean an algorithm A for which there are "bucket boundaries" $b_1 < b_2 < \cdots < b_\ell$ such that A first distributes the n input elements into their rightful buckets, and then sorts each bucket separately, concatenating the results. The degrees of freedom when defining such an algorithm are: (i) the choice of the bucket boundaries; (ii) the method used to distribute input elements to the buckets; and (iii) the method used to sort each bucket. The performance measure COST is the number of comparisons used by the algorithm.[16]

The key steps in the analysis in [ACCL06] can be reinterpreted as proving that this set of bucket-based sorting algorithms has low representation error, in the following sense.

Theorem 3.9 ([ACCL06, Theorem 2.1])**.** *Suppose that each array element a_i is drawn independently from a distribution $\mathcal{D}_i$. Then there exists a bucket-based sorting algorithm with expected running time at most a constant factor times that of the optimal sorting algorithm for $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$.*

The proof in [ACCL06] establishes Theorem 3.9 even when the number ℓ of buckets is only n, each bucket is sorted using InsertionSort, and each element a_i is distributed independently to its rightful

[15] In general, arbitrary local search algorithms can be made β-bounded through time-outs: if such an algorithm always halts within T iterations, then the corresponding assignment rule is T-bounded.

[16] Devroye [Dev86] studies similar families of sorting algorithms, with the goal of characterizing the expected running time as a function of the input distribution.

bucket using a search tree stored in $O(n^c)$ bits, where $c > 0$ is an arbitrary constant (and the running time depends on $\frac{1}{c}$).[17] Let $\mathcal{A}_c$ denote the set of all such bucket-based sorting algorithms.

Theorem 3.9 reduces the task of learning a near-optimal sorting algorithm to the problem of (ϵ, δ)-learning the optimal algorithm from $\mathcal{A}_c$. Corollary 3.4 reduces this learning problem to bounding the pseudo-dimension of $\mathcal{A}_c$. We next prove such a bound, which effectively says that bucket-based sorting algorithms are "relatively simple" algorithms.[18]

Theorem 3.10 (Pseudo-Dimension of Bucket-Based Sorting Algorithms)**.** *The pseudo-dimension of $\mathcal{A}_c$ is $O(n^{1+c})$.*

Proof. Recall from the definitions (Section 3.3.2) that we need to upper bound the size of every set that is shatterable using the bucket-based sorting algorithms in $\mathcal{A}_c$. For us, a set is a fixed set of s inputs (i.e., arrays of length n), $S = x_1, \ldots, x_s$. For a potential witness $r_1, \ldots, r_s \in \mathbb{R}$, every algorithm $A \in \mathcal{A}_c$ induces a binary labeling of each sample x_i, according to whether $\text{COST}(A, x_i)$ is strictly more than or at most r_i. We proceed to bound from above the number of distinct binary labellings of S induced by the algorithms of $\mathcal{A}_c$, for any potential witness.

By definition, an algorithm from $\mathcal{A}_c$ is fully specified by: (i) a choice of n bucket boundaries $b_1 < \cdots < b_n$; and (ii) for each $i = 1, 2, \ldots, n$, a choice of a search tree T_i of size at most $O(n^c)$ for placing x_i in the correct bucket. Call two algorithms $A, A' \in \mathcal{A}_c$ *equivalent* if their sets of bucket boundaries $b_1, \ldots, b_n$ and $b'_1, \ldots, b'_n$ induce the same partition of the sn array elements of the inputs in S — that is, if $x_{ij} < b_k$ if and only if $x_{ij} < b'_k$ (for all i, j, k). The number of equivalence classes of this equivalence relation is at most $\binom{sn+n}{n} \leq (sn+n)^n$. Within an equivalence class, two algorithms that use structurally identical search trees will have identical performance on all s of the samples. Since the search trees of every algorithm of $\mathcal{A}_c$ are described by at most $O(n^{1+c})$ bits, ranging over the algorithms of a single equivalence class generates at most $2^{O(n^{1+c})}$ distinct binary labellings of the s sample inputs. Ranging over all algorithms thus generates at most $(sn+n)^n 2^{O(n^{1+c})}$ labellings. This exceeds 2^s, the requisite number of labellings to shatter S, only if $s = O(n^{1+c})$. □

Theorem 3.10 and Corollary 3.4 imply that $m = \tilde{O}(\frac{H^2}{\epsilon^2} n^{1+c})$ samples are enough to (ϵ, δ)-learn the optimal algorithm in $\mathcal{A}_c$, where H can be taken as the ratio between the maximum and minimum running time of any algorithm in $\mathcal{A}_c$ on any instance.[19] Since the minimum running time is $\Omega(n)$ and we can assume that the maximum running time is $O(n \log n)$ — if an algorithm exceeds this bound, we can abort it and safely run MergeSort instead — we obtain a sample complexity bound of $\tilde{O}(n^{1+c})$.[20]

[17] For small c, each search tree T_i is so small that some searches will go unresolved; such unsuccessful searches are handled by a standard binary search over the buckets.

[18] Not all sorting algorithms are simple in the sense of having polynomial pseudo-dimension. For example, the space lower bound in [ACCL06, Lemma 2.1] can be adapted to show that no class of sorting algorithms with polynomial pseudo-dimension (or fat shattering dimension) has low representation error in the sense of Theorem 3.9 for general distributions over sorting instances, where the array entries need not be independent.

[19] We again use $\tilde{O}(\cdot)$ to suppress logarithmic factors.

[20] In the notation of Theorem 3.2, we are taking $H = \Theta(n \log n)$, $\epsilon = \Theta(n)$, and using the fact that all quantities are $\Omega(n)$ to conclude that all running times are correctly estimated up to a constant factor. The results implicit in [ACCL06] are likewise for relative error.

Remark 3.11 (Comparison to [ACCL06]). *The sample complexity bound implicit in [ACCL06] for learning a near-optimal sorting algorithm is $\tilde{O}(n^c)$, a linear factor better than the $\tilde{O}(n^{1+c})$ bound implied by Theorem 3.10. There is good reason for this: the pseudo-dimension bound of Theorem 3.10 implies that an even harder problem has sample complexity $\tilde{O}(n^{1+c})$, namely that of learning a near-optimal bucket-based sorting algorithm with respect to an* arbitrary *distribution over inputs,* even with correlated array elements.[21] *The bound of $\tilde{O}(n^c)$ in [ACCL06] applies only to the problem of learning a near-optimal bucket-based sorting algorithm for an unknown input distribution with independent array entries — the savings comes from the fact that all n near-optimal search trees $T_1, \ldots, T_n$ can be learned in parallel.*

3.3.5 Application: Feature-Based Algorithm Selection

Previous sections studied the problem of selecting a single algorithm for use in an application domain — of using training data to make an informed commitment to a single algorithm from a class $\mathcal{A}$, which is then used on all future instances. A more refined and ambitious approach is to select an algorithm based both on previous experience *and on the current instance to be solved.* This approach assumes, as in the scenario in Section 3.2.4, that it is feasible to quickly compute some features of an instance and then to select an algorithm as a function of these features.

Throughout this section, we augment the basic model of Section 3.3.1 with:

5. A set $\mathcal{F}$ of possible instance feature values, and a map $f : X \to \mathcal{F}$ that computes the features of a given instance.[22]

For instance, if X is the set of SAT instances, then $f(x)$ might encode the clause/variable ratio of the instance x, Knuth's estimate of the search tree size [Knu75], and so on.

When the set $\mathcal{F}$ of possible instance feature values is finite, the guarantees for the basic model extend easily with a linear (in $|\mathcal{F}|$) degradation in the pseudo-dimension.[23] To explain, we add an additional ingredient to the model.

6. A set $\mathcal{G}$ of *algorithm selection maps*, with each $g \in \mathcal{G}$ a function from $\mathcal{F}$ to $\mathcal{A}$.

An algorithm selection map recommends an algorithm as a function of the features of an instance.

We can view an algorithm selection map g as a real-valued function defined on the instance space X, with $g(x)$ defined as $\textsc{cost}(g(f(x)), x)$. That is, $g(x)$ is the running time on x of the algorithm $g(f(x))$ advocated by g, given that x has features $f(x)$. The basic model studied earlier is the special case where $\mathcal{G}$ is the set of constant functions, which are in correspondence with the algorithms of $\mathcal{A}$.

[21] When array elements are not independent, however, Theorem 3.9 fails and the best bucket-based sorting algorithm might be more than a constant-factor worse than the optimal sorting algorithm.

[22] Defining a good feature set is a notoriously challenging and important problem, but it is beyond the scope of our model — we take the set $\mathcal{F}$ and map f as given.

[23] For example, [XHHL08] first predicts whether or not a given SAT instance is satisfiable or not, and then uses a "conditional" empirical performance model to choose a SAT solver. This can be viewed as an example with $|\mathcal{F}| = 2$, corresponding to the feature values "looks satisfiable" and "looks unsatisfiable."

Corollary 3.4 reduces bounding the sample complexity of (ϵ, δ)-learning the best algorithm selection map of $\mathcal{G}$ to bounding the pseudo-dimension of the set of real-valued functions induced by $\mathcal{G}$. When $\mathcal{G}$ is finite, there is a trivial upper bound of $\log_2 |\mathcal{G}|$. The pseudo-dimension is also small whenever $\mathcal{F}$ is small and the set $\mathcal{A}$ of algorithms has small pseudo-dimension.[24]

Proposition 3.12 (Pseudo-Dimension of Algorithm Selection Maps)**.** *If $\mathcal{G}$ is a set of algorithm selection maps from a finite set $\mathcal{F}$ to a set $\mathcal{A}$ of algorithms with pseudo-dimension d, then $\mathcal{G}$ has pseudo-dimension at most $|\mathcal{F}|d$.*

Proof. A set of inputs of size $|\mathcal{F}|d + 1$ is shattered only if there is a shattered set of inputs with identical features of size $d + 1$. □

Now suppose $\mathcal{F}$ is very large (or infinite). We focus on the case where $\mathcal{A}$ is small enough that it is feasible to learn a separate performance prediction model for each algorithm $A \in \mathcal{A}$ (though see Remark 3.15). This is exactly the approach taken in the motivating example of empirical performance models (EPMs) for SAT described in Section 3.2.4. In this case, we augment the basic model to include a family of performance predictors.

6. A set $\mathcal{P}$ of *performance predictors*, with each $p \in \mathcal{P}$ a function from $\mathcal{F}$ to $\mathbb{R}$.

Performance predictors play the same role as the EPMs used in [XHHL08].

The goal is to learn, for each algorithm $A \in \mathcal{A}$, among all permitted predictors $p \in \mathcal{P}$, the one that minimizes some loss function. Like the performance measure COST, we take this loss function as given. The most commonly used loss function is squared error; in this case, for each $A \in \mathcal{A}$ we aim to compute the function that minimizes

$$\mathbf{E}_{x \sim \mathcal{D}}\left[(\textsc{cost}(A, x) - p(f(x)))^2\right]$$

over $p \in \mathcal{P}$.[25] For a fixed algorithm A, this is a standard regression problem, with domain $\mathcal{F}$, real-valued labels, and a distribution on $\mathcal{F} \times \mathbb{R}$ induced by $\mathcal{D}$ via $x \mapsto (f(x), \textsc{cost}(A, x))$. Bounding the sample complexity of this learning problem reduces to bounding the pseudo-dimension of $\mathcal{P}$. For standard choices of $\mathcal{P}$, such bounds are well known. For example, suppose the set $\mathcal{P}$ is the class of *linear predictors*, with each $p \in \mathcal{P}$ having the form $p(f(x)) = a^T f(x)$ for some coefficient vector $a \in \mathbb{R}^d$.[26] The following is well known (see e.g. [AB99]).

Proposition 3.13 (Pseudo-Dimension of Linear Predictors)**.** *If $\mathcal{F}$ contains real-valued d-dimensional features and $\mathcal{P}$ is the set of linear predictors, then the pseudo-dimension of $\mathcal{P}$ is at most d.*

[24] When $\mathcal{G}$ is the set of all maps from $\mathcal{F}$ to $\mathcal{A}$ and every feature value of $\mathcal{F}$ appears with approximately the same probability, one can alternatively just separately learn the best algorithm for each feature value.

[25] Note that the expected loss incurred by the best predictor depends on the choices of the predictor set $\mathcal{P}$, the feature set $\mathcal{F}$, and map f. Again, these choices are outside our model.

[26] A linear model might sound unreasonably simple for the task of predicting the running time of an algorithm, but significant complexity can be included in the feature map $f(x)$. For example, each coordinate of $f(x)$ could be a nonlinear combination of several "basic features" of x. Indeed, linear models often exhibit surprisingly good empirical performance, given a judicious choice of a feature set [LNS09].

If all functions in $\mathcal{P}$ map all possible φ to $[0, H]$, then Proposition 3.13 and Corollary 3.4 imply a sample complexity bound of $\tilde{O}(\frac{H^4}{\epsilon^2} d)$ for (ϵ, δ)-learning the predictor with minimum expected square error. Similar results hold, with worse dependence on d, if $\mathcal{P}$ is a set of low-degree polynomials [AB99].

For another example, suppose $\mathcal{P}_\ell$ is the set of regression trees with at most ℓ nodes, where each internal node performs an inequality test on a coordinate of the feature vector φ (and leaves are labeled with performance estimates).[27] This class also has low pseudo-dimension[28], and hence the problem of learning a near-optimal predictor has correspondingly small sample complexity.

Proposition 3.14 (Pseudo-Dimension of Regression Trees)**.** *Suppose $\mathcal{F}$ contains real-valued d-dimensional features and let $\mathcal{P}_\ell$ be the set of regression trees with at most ℓ nodes, where each node performs an inequality test on one of the features. Then, the pseudo-dimension of $\mathcal{P}_\ell$ is $O(\ell \log(\ell d))$.*

Remark 3.15 (Extension to Large $\mathcal{A}$)**.** *We can also extend our approach to scenarios with a large or infinite set $\mathcal{A}$ of possible algorithms. This extension is relevant to state-of-the-art empirical approaches to the auto-tuning of algorithms with many parameters, such as mathematical programming solvers [HXHL14]; see also the discussion in Section 3.2.3. (Instantiating all of the parameters yields a fixed algorithm; ranging over all possible parameter values yields the set $\mathcal{A}$.) Analogous to our formalism for accommodating a large number of possible features, we now assume that there is a set $\mathcal{F}'$ of possible "algorithm feature values" and a mapping f' that computes the features of a given algorithm. A performance predictor is now a map from $\mathcal{F} \times \mathcal{F}'$ to $\mathbb{R}$, taking as input the features of an algorithm A and of an instance x, and returning as output an estimate of A's performance on x. If $\mathcal{P}$ is the set of linear predictors, for example, then by Proposition 3.13 its pseudo-dimension is $d + d'$, where d and d' denote the dimensions of $\mathcal{F}$ and $\mathcal{F}'$, respectively.*

3.3.6 Application: Choosing the Step Size in Gradient Descent

For our last PAC example, we give sample complexity results for the problem of choosing the best step size in gradient descent. When gradient descent is used in practice, the step size is generally taken much larger than the upper limits suggested by theoretical guarantees, and often converges in many fewer iterations than with the step size suggested by theory. This motivates the problem of learning the step size from examples. We view this as a baby step towards reasoning more generally about the problem of learning good parameters for machine learning algorithms.

In this section, we look at a setting where the approximation quality is fixed for all algorithms, and the performance measure COST is related to the running time of the algorithm. Unlike the applications we've seen so far, the parameter space here satisfies a Lipschitz-like condition, and we can follow the discretization approach suggested in Remark 3.7.

[27] Regression trees, and random forests thereof, have emerged as a popular class of predictors in empirical work on application-specific algorithm selection [HXHL14].

[28] We suspect this fact is known, but have been unable to locate a suitable reference.

Gradient Descent Preliminaries

Recall the basic gradient descent algorithm for minimizing a function f given an initial point z_0 over $\mathbb{R}^n$:

1. Initialize $z := z_0$.
2. While $\|\nabla f(z)\|_2 > \nu$:
   (a) $z := z - \rho \cdot \nabla f(z)$.

We take the error tolerance ν as given and focus on the more interesting parameter, the step size ρ. Bigger values of ρ have the potential to make more progress in each step, but run the risk of overshooting a minimum of f.

We instantiate the basic model (Section 3.3.1) to study the problem of learning the best step size. There is an unknown distribution $\mathcal{D}$ over instances, where an instance $x \in \Pi$ consists of a function f and an initial point z_0. Each algorithm A_ρ of $\mathcal{A}$ is the basic gradient descent algorithm above, with some choice ρ of a step size drawn from some fixed interval $[\rho_\ell, \rho_u] \subset (0, \infty)$. The performance measure $\text{COST}(A, x)$ is the number of iterations (i.e., steps) taken by the algorithm for the instance x.

To obtain positive results, we need to restrict the allowable functions f (see the end of Section 3.3.6). First, we assume that every function f is convex and L-smooth for a known L. A function f is *L-smooth* if it is everywhere differentiable, and $\|\nabla f(z_1) - \nabla f(z_2)\| \le L\|z_1 - z_2\|$ for all z_1 and z_2 (all norms in this section are the ℓ_2 norm). Since gradient descent is translation invariant, and f is convex, we can assume for convenience that the (uniquely attained) minimum value of f is 0, with $f(0) = 0$.

Second, we assume that the magnitudes of the initial points are bounded, with $\|z_0\| \le Z$ for some known constant $Z > \nu$.

Third, we assume that there is a known constant $c \in (0, 1)$ such that $\|z - \rho\nabla f(z)\| \le (1 - c)\|z\|$ for all $\rho \in [\rho_\ell, \rho_u]$. In other words, the norm of any point z — equivalently, the distance to the global minimum — decreases by some minimum factor after each gradient descent step. We refer to this as the *guaranteed progress* condition. This is satisfied (for instance) by L-smooth, m-strongly convex functions[29], which is a well studied regime (see e.g. [BV04]). The standard analysis of gradient descent implies that $c \ge \rho m$ for $\rho \le 2/(m + L)$ over this class of functions.

Under these restrictions, we will be able to compute a nearly optimal ρ given a reasonable number of samples from $\mathcal{D}$.

Other Notation All norms in this section are ℓ_2-norms. Unless otherwise stated, ρ means ρ restricted to $[\rho_\ell, \rho_u]$, and z means z such that $\|z\| \le Z$. We let $g(z, \rho) := z - \rho\nabla f(z)$ be the result of taking a single gradient descent step, and $g^j(z, \rho)$ be the result of taking j gradient descent steps.

[29] A (continuously differentiable) function f is *m-strongly convex* if $f(y) \ge f(w) + \nabla f(w)^T(y - w) + \frac{m}{2}\|y - w\|^2$ for all $w, y \in \mathbb{R}^n$. The usual notion of convexity is the same as 0-strong convexity. Note that the definition of L-smooth implies $m \le L$.

Typical textbook treatments of gradient descent assume $\rho < 2/L$ or $\rho \le 2/(m+L)$, which give various convergence and running time guarantees. The learning results of this section apply for any ρ, but this natural threshold will still appear in our analysis and results. Let $D(\rho) := \max\{1, L\rho - 1\}$ denote how far ρ is from $2/L$.

By the guaranteed progress condition, $\|g^j(z,\rho)\| \le (1-c)^j \|z\|$, and so by L-smoothness,

$$\|\nabla f(g^j(z,\rho))\| \le (1-c)^j L \|z\|.$$

Since $\|z_0\| \le Z$, and we stop once the gradient is $\le \nu$, $\textsc{cost}(A_\rho, x) \le \log(\nu/LZ)/\log(1-c)$ for all ρ and x. Let $H = \log(\nu/LZ)/\log(1-c)$.

A Lipschitz-like Bound on $\textsc{cost}(A_\rho, x)$ as a Function of ρ.

This will be the bulk of the argument. Our first lemma shows that for fixed ρ, the gradient descent step g is a Lipschitz function of z, even when ρ is larger than $2/L$. One might hope that the guaranteed progress condition would be enough to show that (say) g is a contraction, but the Lipschitzness of g actually comes from the L-smoothness. (It is not too hard to come up with non-smooth functions that make guaranteed progress, and where g is arbitrarily non-Lipschitz.)

Lemma 3.16. $\|g(w,\rho) - g(y,\rho)\| \le D(\rho)\|w - y\|$.

Proof. For notational simplicity, let $\alpha = \|w - y\|$ and $\beta = \|\nabla f(w) - \nabla f(y)\|$. Now,

$$\begin{aligned}
\|g(w,\rho) - g(y,\rho)\|^2 &= \|(w-y) - \rho(\nabla f(w) - \nabla f(y))\|^2 \\
&= \alpha^2 + \rho^2\beta^2 - 2\rho\langle \alpha, \beta\rangle \\
&\le \alpha^2 + \rho^2\beta^2 - 2\rho\beta^2/L \\
&= \alpha^2 + \beta^2\rho(\rho - 2/L).
\end{aligned}$$

The only inequality above is a restatement of a property of L-smooth functions called the co-coercivity of the gradient, namely that $\langle \alpha, \beta\rangle \ge \beta^2/L$.

Now, if $\rho \le 2/L$, then $\rho(\rho - 2/L) \le 0$, and we're done. Otherwise, L-smoothness implies $\beta \le L\alpha$, so the above is at most $\alpha^2(1 + L\rho(L\rho - 2))$, which is the desired result. □

The next lemma bounds how far two gradient descent paths can drift from each other, if they start at the same point. The main thing to note is that the right hand side goes to 0 as η becomes close to ρ.

Lemma 3.17. *For any z, j, and $\rho \le \eta$,*

$$\|g^j(z,\rho) - g^j(z,\eta)\| \le (\eta - \rho)\frac{D(\rho)^j LZ}{c}.$$

Proof. We first bound $\|g(w,\rho)-g(y,\eta)\|$, for any w and y. We have

$$g(w,\rho)-g(y,\eta)=[w-\rho\nabla f(w)]-[y-\eta\nabla f(y)]=g(w,\rho)-[g(y,\rho)-(\eta-\rho)\nabla f(y)]$$

by definition of g. The triangle inequality and Lemma 3.16 then give

$$\|g(w,\rho)-g(y,\eta)\|=\|g(w,\rho)-g(y,\rho)+(\eta-\rho)\nabla f(y)\|\leq D(\rho)\,\|w-y\|+(\eta-\rho)\|\nabla f(y)\|.$$

Plugging in $w=g^j(z,\rho)$ and $y=g^j(z,\eta)$, we have

$$\|g^{j+1}(z,\rho)-g^{j+1}(z,\eta)\|\leq D(\rho)\,\|g^j(z,\rho)-g^j(z,\eta)\|+(\eta-\rho)\|\nabla f(g^j(z,\eta))\|$$

for all j.

Now,

$$\|\nabla f(g^j(z,\eta))\|\leq L\|g^j(z,\eta)\|\leq L\|z\|(1-c)^j\leq LZ(1-c)^j,$$

where the first inequality is from L-smoothness, and the second is from the guaranteed progress condition. Letting $r_j=\|g^j(z,\rho)-g^j(z,\eta)\|$, we now have the simple recurrence $r_0=0$, and $r_{j+1}\leq D(\rho)\,r_j+(\eta-\rho)LZ(1-c)^j$. One can check via induction that

$$r_{j+1}\leq D(\rho)^j(\eta-\rho)LZ\sum_{i=0}^{j}(1-c)^iD(\rho)^{-i}$$

for all j. Recall that $D(\rho)\geq 1$. Rounding $D(\rho)^{-i}$ up to 1 and doing the summation gives the desired result. □

Finally, we show that $\textsc{cost}(A_\rho,x)$ is essentially Lipschitz in ρ. The "essentially" is necessary, since $\textsc{cost}$ is integer-valued.

Lemma 3.18. $|\textsc{cost}(A_\rho,x)-\textsc{cost}(A_\eta,x)|\leq 1$ *for all* x*,* ρ*, and* η *with* $0\leq\eta-\rho\leq\frac{\nu c^2}{LZ}D(\rho)^{-H}$*.*

Proof. Assume that $\textsc{cost}(A_\eta,x)\leq\textsc{cost}(A_\rho,x)$; the argument in the other case is similar. Let $j=\textsc{cost}(A_\eta,x)$, and recall that $j\leq H$. By Lemma 3.17, $\|g^j(x,\rho)-g^j(x,\eta)\|\leq\nu c$. Hence, by the triangle inequality,

$$\|g^j(x,\rho)\|\leq\nu c+\|g^j(x,\eta)\|\leq\nu c+\nu.$$

Now, by the guaranteed progress condition, $\|w\|-\|g(w,p)\|\geq c\|w\|$ for all w. Since we only run a gradient descent step on w if $\|w\|>\nu$, each step of gradient descent run by any algorithm in $\mathcal{A}$ drops the magnitude of w by at least νc.

Setting $w=g^j(x,\rho)$, we see that either $\|g^j(x,\rho)\|\leq\nu$, and $\textsc{cost}(A_\rho,x)=j$, or that $\|g^{j+1}(x,\rho)\|\leq(\nu c+\nu)-\nu c=\nu$, and $\textsc{cost}(A_\rho,x)=j+1$, as desired. □

Learning the Best Step Size

We can now apply the discretization approach suggested by Remark 3.7. Let $K = \frac{\nu c^2}{LZ} D(\rho_u)^{-H}$. Note that since D is an increasing function, K is less than or equal to the $\frac{\nu c^2}{LZ} D(\rho)^{-H}$ of Lemma 3.18 for every ρ. Let N be a minimal K-net, such as all integer multiples of K that lie in $[\rho_\ell, \rho_u]$. Note that $|N| \leq \rho_u/K + 1$.

We tie everything together in the theorem below.[30]

Theorem 3.19 (Learnability of Step Size in Gradient Descent)**.** *There is a learning algorithm that* $(1+\epsilon, \delta)$*-learns the optimal algorithm in* $\mathcal{A}$ *using* $m = \tilde{O}(H^3/\epsilon^2)$ *samples from* $\mathcal{D}$*.*[31]

Proof. The pseudo-dimension of $\mathcal{A}_N = \{A_\rho : \rho \in N\}$ is at most $\log |N|$, since $\mathcal{A}_N$ is a finite set. Since $\mathcal{A}_N$ is finite, it also trivially admits an ERM algorithm L_N, and Corollary 3.4 implies that L_N (ϵ, δ)-learns the optimal algorithm in $\mathcal{A}_N$ using $m = \tilde{O}(H^2 \log |N| / \epsilon^2)$ samples.

Now, Lemma 3.18 implies that for every ρ, there is a $\eta \in N$ such that, for every distribution $\mathcal{D}$, the difference in expected costs of A_η and A_ρ is at most 1. Thus L_N $(1+\epsilon, \delta)$-learns the optimal algorithm in $\mathcal{A}$ using $m = \tilde{O}(H^2 \epsilon^{-2} \log |N|)$ samples.

Since $\log |N| = \tilde{O}(H)$, we get the desired result. □

A Hard Example for Gradient Descent

We now depict a family $\mathcal{F}$ of functions for which $\mathcal{A}$ has arbitrarily high pseudo-dimension, justifying the need to restrict the set of allowable functions above. For each member $f \in \mathcal{F}$, $f : \mathbb{R}^2 \to \mathbb{R}$, and we parameterize $f_I \in \mathcal{F}$ by finite subsets $I \subset [0,1]$. An aerial view of f_I appears in Figure 3.1.

The "squiggle" $s(I)$ intersects the relevant axis at exactly I (to be concrete, let $s(I)$ be the monic polynomial with roots at I). We fix the initial point z_0 to be at the tail of the arrow for all instances, and fix ρ_ℓ and ρ_u so that the first step of gradient descent takes z_0 from the red incline into the middle of the black and blue area. Let x_I be the instance corresponding to f_I with starting point z_0. If for a certain ρ and I, $g(z_0, \rho)$ lands in the flat, black area, gradient descent stops immediately and $\textsc{cost}(A_\rho, x_I) = 1$. If $g(z_0, \rho)$ instead lands in the sloped, blue area, $\textsc{cost}(A_\rho, x_I) \gg 1$.

It should be clear that $\mathcal{F}$ can shatter any finite subset of (ρ_ℓ, ρ_u), and hence has arbitrarily large pseudo-dimension. One can also make slight modifications to ensure that all the functions in $\mathcal{F}$ are continuously differentiable and L-smooth.

[30] Alternatively, this guarantee can be phrased in terms of the fat-shattering dimension (see e.g. [AB99]). In particular, $\mathcal{A}$ has 1.001-fat shattering dimension at most $\log |N| = \tilde{O}(H)$.

[31] We use $\tilde{O}(\cdot)$ to suppress logarithmic factors in $Z/\nu, c, L$ and ρ_u.

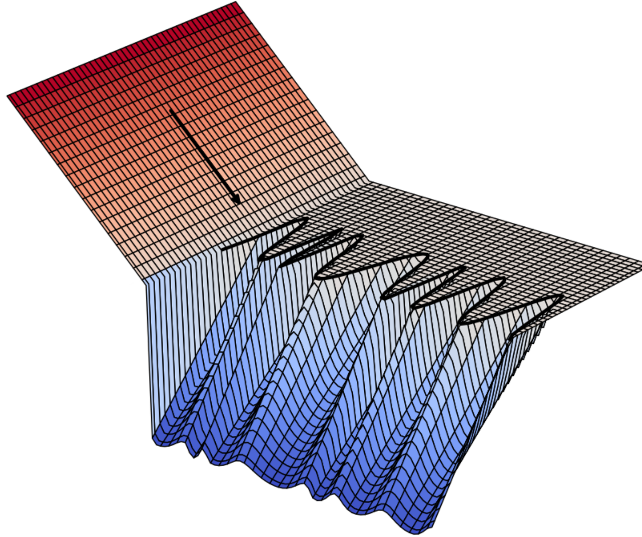

Figure 3.1: A family of functions for which gradient descent has arbitrarily high pseudo-dimension.

3.4 Online Learning of Application-Specific Algorithms

This section studies the problem of learning the best application-specific algorithm *online*, with instances arriving one-by-one.[32] The goal is choose an algorithm at each time step, before seeing the next instance, so that the average performance is close to that of the best fixed algorithm in hindsight. This contrasts with the statistical (or "batch") learning setup used in Section 3.3, where the goal was to identify a single algorithm from a batch of training instances that generalizes well to future instances from the same distribution. For many of the motivating examples in Section 3.2, both the statistical and online learning approaches are relevant. The distribution-free online learning formalism of this section may be particularly appropriate when instances cannot be modeled as i.i.d. draws from an unknown distribution.

3.4.1 The Online Learning Model

Our online learning model shares with the basic model of Section 3.3.1 a computational or optimization problem Π (e.g., MWIS), a set $\mathcal{A}$ of algorithms for Π (e.g., a single-parameter family of greedy heuristics), and a performance measure $\textsc{cost} : \mathcal{A} \times \Pi \to [0, 1]$ (e.g., the total weight of the returned solution).[33] Rather than modeling the specifics of an application domain via an unknown distribution $\mathcal{D}$ over instances, however, we use an unknown instance *sequence* $x_1, \ldots, x_T$.[34]

[32]The online model is obviously relevant when training data arrives over time. Also, even with offline data sets that are very large, it can be computationally necessary to process training data in a one-pass, online fashion.

[33]One could also have $\textsc{cost}$ take values in $[0, H]$ rather than $[0, 1]$, to parallel the PAC setting; we set $H = 1$ here since the dependence on H will not be interesting.

[34]For simplicity, we assume that the time horizon T is known. This assumption can be removed by standard doubling techniques (e.g. [CL06]).

A learning algorithm now outputs a sequence $A_1, \ldots, A_T$ of algorithms, rather than a single algorithm. Each algorithm A_i is chosen (perhaps probabilistically) with knowledge only of the previous instances $x_1, \ldots, x_{i-1}$. The standard goal in online learning is to choose $A_1, \ldots, A_T$ to minimize the worst-case (over $x_1, \ldots, x_T$) *regret*, defined as the average performance loss relative to the best algorithm $A \in \mathcal{A}$ in hindsight:[35]

$$\frac{1}{T}\left(\sup_{A \in \mathcal{A}} \sum_{t=1}^{T} \text{COST}(A, x_i) - \sum_{t=1}^{T} \text{COST}(A_i, x_i)\right). \tag{3.2}$$

A *no-regret* learning algorithm has expected (over its coin tosses) regret $o(1)$, as $T \to \infty$, for every instance sequence. The design and analysis of no-regret online learning algorithms is a mature field (see e.g. [CL06]). For example, many no-regret online learning algorithms are known for the case of a finite set $|\mathcal{A}|$ (such as the "multiplicative weights" algorithm).

3.4.2 An Impossibility Result for Worst-Case Instances

This section proves an impossibility result for no-regret online learning algorithms for the problem of application-specific algorithm selection. We show this for the running example in Section 3.3.3: maximum-weight independent set (MWIS) heuristics[36] that, for some parameter $\rho \in [0, 1]$, process the vertices in order of nonincreasing value of $w_v/(1 + \deg(v))^\rho$. Let $\mathcal{A}$ denote the set of all such MWIS algorithms. Since $\mathcal{A}$ is an infinite set, standard no-regret results (for a finite number of actions) do not immediately apply. In online learning, infinite sets of options are normally controlled through a Lipschitz condition, stating that "nearby" actions always yield approximately the same performance; our set $\mathcal{A}$ does not possess such a Lipschitz property (recall Remark 3.7). The next section shows that these issues are not mere technicalities — there is enough complexity in the set $\mathcal{A}$ of MWIS heuristics to preclude a no-regret learning algorithm.

A Hard Example for MWIS

We show a distribution over sequences of MWIS instances for which every (possibly randomized) algorithm has expected regret $1 - o_n(1)$. Here and for the rest of this section, by $o_n(1)$ we mean a function that is independent of T and tends to 0 as the number of vertices n tends to infinity. Recall that $\text{COST}(A_\rho, x)$ is the total weight of the returned independent set, and we are trying to maximize this quantity. The key construction is the following:

Lemma 3.20. *For any constants $0 < r < s < 1$, there exists a MWIS instance x on at most n vertices such that $\text{COST}(A_\rho, x) = 1$ when $\rho \in (r, s)$, and $\text{COST}(A_\rho, x) = o_n(1)$ when $\rho < r$ or $\rho > s$.*

Proof. Let A, B, and C be 3 sets of vertices of sizes $m^2 - 2, m^3 - 1$, and $m^2 + m + 1$ respectively, such that their sum $m^3 + 2m^2 + m$ is between $n/2$ and n. Let (A, B) be a complete bipartite graph.

[35] Without loss of generality, we assume COST corresponds to a maximization objective.

[36] Section 3.3.3 defined adaptive and non-adaptive versions of the MWIS heuristic. All of the results in Section 3.4 apply to both, so we usually won't distinguish between them.

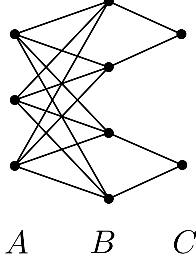

Figure 3.2: A rough depiction of the MWIS example from Lemma 3.20.

	size	weight	deg	weight/(deg+1)$^\rho$	size × weight
A	m^2-2	tm^r	m^3-1	$tm^{r-3\rho}$	$o_n(1)$
B	m^3-1	t	m^2-1	$tm^{-2\rho}$	1
C	m^2+m+1	tm^{-s}	$m-1$	$tm^{-s-\rho}$	$o_n(1)$

Figure 3.3: Details and simple calculations for the vertex sets comprising the MWIS example from Lemma 3.20.

Let (B, C) also be a bipartite graph, with each vertex of B connected to exactly one vertex of C, and each vertex of C connected to exactly $m-1$ vertices of B. See Figure 3.2.

Now, set the weight of every vertex in A, B, and C to tm^r, t, and tm^{-s}), respectively, for $t = (m^3-1)^{-1}$. Figure 3.3 summarizes some straightforward calculations. We now calculate the cost of A_ρ on this instance.

If $\rho < r$, the algorithm A_ρ first chooses a vertex in A, which immediately removes all of B, leaving at most A and C in the independent set. The total weight of A and C is $o_n(1)$, so $\textsc{cost}(A_\rho)$ is $o_n(1)$.

If $\rho > s$, the algorithm first chooses a vertex in C, which removes a small chunk of B. In the non-adaptive setting, A_ρ simply continues choosing vertices of C until B is gone. In the adaptive setting, the degrees of the remaining elements of B never change, but the degrees of A decrease as we pick more and more elements of C. We eventually pick a vertex of A, which immediately removes the rest of B. In either case, the returned independent set has no elements from B, and hence has $\textsc{cost}$ $o_n(1)$.

If $\rho \in (r, s)$, the algorithm first picks a vertex of B, immediately removing all of A, and one element of C. The remaining graph comprises $m-2$ isolated vertices of B (which get added to the independent set), and m^2+m stars with centers in C and leaves in B. It is easy to see that both the adaptive and the non-adaptive versions of the heuristic return exactly B. □

We are now ready to state the main result of this section.

Theorem 3.21 (Impossibility of Worst-Case Online Learning)**.** *There is a distribution on MWIS input sequences over which every algorithm has expected regret* $1 - o_n(1)$.

Proof. Let $t_j = (r_j, s_j)$ be a distribution over sequences of nested intervals with $s_j - r_j = n^{-j}$, $t_0 = (0,1)$, and with t_j chosen uniformly at random from within t_{j-1}. Let x_j be an MWIS instance on up to n vertices such that $\textsc{cost}(A_\rho, x) = 1$ for $\rho \in (r_j, s_j)$, and $\textsc{cost}(A_\rho, x) = o_n(1)$ for $\rho < r_j$ and $\rho > s_j$ (Lemma 3.20).

The adversary presents the instances $x_1, x_2, \ldots, x_T$, in that order. For every $\rho \in t_T$, $\textsc{cost}(A_\rho, x_j) = 1$ for all j. However, at every step t, no algorithm can have a better than $1/n$ chance of picking a ρ_t for which $\textsc{cost}(A_{\rho_t}, x_t) = \Theta_n(1)$, even given $x_1, x_2, \ldots, x_{t-1}$ and full knowledge of how the sequence is generated. ☐

3.4.3 A Smoothed Analysis

Despite the negative result above, we can show a "low-regret" learning algorithm for MWIS under a slight restriction on how the instances x_t are chosen. By low-regret we mean that the regret can be made polynomially small as a function of the number of vertices n. This is not the same as the no-regret condition, which requires regret tending to 0 as $T \to \infty$. Nevertheless, inverse polynomially small regret $\text{poly}(n^{-1})$ is a huge improvement over the constant regret incurred in the worst-case lower bound (Theorem 3.21). Let $\text{poly}(n^{-1})$ denote a function that is independent of$T$, and which can be taken as an arbitrarily small polynomial in n^{-1}.

We take the approach suggested by smoothed analysis [ST09]. Fix a parameter $\sigma \in (0,1)$. We allow each MWIS instance x_t to have an arbitrary graph on n vertices, but we replace each vertex weight w_v with a probability distribution $\Delta_{t,v}$ with density at most σ^{-1} (pointwise) and support in $[0,1]$. A simple example of such a distribution with $\sigma = 0.1$ is the uniform distribution on $[0.6, 0.65] \cup [0.82, 0.87]$. To instantiate the instance x_t, we draw each vertex weight from its distribution $\Delta_{t,v}$. We call such an instance a *σ-smooth MWIS instance.*

For small σ, this is quite a weak restriction. As $\sigma \to 0$ we return to the worst-case setting, and Theorem 3.21 can be extended to the case of σ exponentially small in n. Here, we think of σ as bounded below by an (arbitrarily small) inverse polynomial function of n. One example of such a smoothing is to start with an arbitrary MWIS instance, keep the first $O(\log n)$ bits of every weight, and set the remaining lower-order bits at random.

The main result of this section is a polynomial-time low-regret learning algorithm for sequences of σ-smooth MWIS instances. Our strategy is to take a finite net $N \subset [0,1]$ such that, for every algorithm A_ρ and smoothed instance x_t, with high probability over x_t the performance of A_ρ is identical to that of some algorithm in $\{A_\eta : \eta \in N\}$. We can then use any off-the-shelf no-regret algorithm to output a sequence of algorithms from the finite set $\{A_\eta : \eta \in N\}$, and show the desired regret bound.

A Low-Regret Algorithm for σ-Smooth MWIS

We start with some definitions. For a fixed x, let $\tau'(x)$ be the set of *transition points*, namely,[37]

$$\tau'(x) := \{\rho : A_{\rho-\omega}(x) \neq A_{\rho+\omega}(x) \text{ for arbitrarily small } \omega\}.$$

It is easy to see $\tau'(x) \subset \tau(x)$, where

$$\tau(x) := \{\rho : w_{v_1}/k_1^\rho = w_{v_2}/k_2^\rho \text{ for some } v_1, v_2, k_1, k_2 \in [n];\ k_1, k_2 \geq 2\}.$$

With probability 1, the vertex weights w_v are all distinct and non-zero, so we can rewrite τ as

$$\tau(x) := \{\rho(v_1, v_2, k_1, k_2) \ :\ v_1, v_2, k_1, k_2 \in [n];\ k_1, k_2 \geq 2;\ k_1 \neq k_2\},$$

where

$$\rho(v_1, v_2, k_1, k_2) = \frac{\ln(w_{v_1}) - \ln(w_{v_2})}{\ln(k_1) - \ln(k_2)} \tag{3.3}$$

and ln is the natural logarithm function. The main technical task is to show that no two elements of $\tau(x_1) \cup \cdots \cup \tau(x_m)$ are within q of each other, for a sufficiently large q and sufficiently large m, and with high enough probability over the randomness in the weights of the x_t's.

We first make a few straightforward computations. The following brings the noise into log space.

Lemma 3.22. *If X is a random variable over $(0, 1]$ with density at most δ, then $\ln(X)$ also has density at most δ.*

Proof. Let $Y = \ln(X)$, let $f(x)$ be the density of X at x, and let $g(y)$ be the density of Y at y. Note that $X = e^Y$, and let $v(y) = e^y$. Then $g(y) = f(v(y)) \cdot v'(y) \leq f(v(y)) \leq \delta$ for all y. □

Since $|\ln(k_1) - \ln(k_2)| \leq \ln n$, Lemma 3.22 and our definition of σ-smoothness implies the following.

Corollary 3.23. *For every σ-smooth MWIS instance x, and every $v_1, v_2, k_1, k_2 \in [n]$, $k_1, k_2 \geq 2$, $k_1 \neq k_2$, the density of $\rho(v_1, v_2, k_1, k_2)$ is bounded by $\sigma^{-1} \ln n$.*

We now show that it is unlikely that two distinct elements of $\tau(x_1) \cup \cdots \cup \tau(x_m)$ are very close to each other.

Lemma 3.24. *Let $x_1, \ldots, x_m$ be σ-smooth MWIS instances. The probability that no two distinct elements of $\tau(x_1) \cup \cdots \cup \tau(x_m)$ are within q of each other is at least $1 - 4q\sigma^{-1}m^2n^8 \ln n$.*

Proof. Fix instances x and x', and choices of (v_1, v_2, k_1, k_2) and (v_1', v_2', k_1', k_2'). Denote by ρ and ρ' the corresponding random variables, defined as in (3.3). We compute the probability that $|\rho - \rho'| \leq q$ under various scenarios, over the randomness in the vertex weights. We can ignore the case where $x = x'$, $v_1 = v_1'$, $v_2 = v_2'$, and $k_1/k_2 = k_1'/k_2'$, since then $\rho = \rho'$ with probability 1. We consider three other cases.

[37] The corner cases $\rho = 0$ and $\rho = 1$ require straightforward but wordy special handling in this statement and in several others in this section. We omit these details to keep the argument free of clutter.

Case 1: Suppose $x \neq x'$, and/or $\{v_1, v_2\}$ and $\{v_1', v_2'\}$ don't intersect. In this case, ρ and ρ' are independent random variables. Hence the maximum density of $\rho - \rho'$ is at most the maximum density of ρ, which is $\sigma^{-1} \ln n$ by Corollary 3.23. The probability that $|\rho - \rho'| \leq q$ is hence at most $2q \cdot \sigma^{-1} \ln n$.

Case 2: Suppose $x = x'$, and $\{v_1, v_2\}$ and $\{v_1', v_2'\}$ share exactly one element, say $v_2 = v_2'$. Then $\rho - \rho'$ has the form $X - Y$, where $X = \frac{\ln(w_{v_1})}{\ln(k_1) - \ln(k_2)}$ and X and Y are independent. Since the maximum density of X is at most $\sigma^{-1} \ln n$ (by Lemma 3.22), the probability that $|\rho - \rho'| \leq q$ is again at most $2q \cdot \sigma^{-1} \ln n$.

Case 3: Suppose $x = x'$ and $\{v_1, v_2\} = \{v_1', v_2'\}$. In this case, $k_1/k_2 \neq k_1'/k_2'$. Then

$$
\begin{aligned}
|\rho - \rho'| &= \left| \big(\ln(w_{v_1}) - \ln(w_{v_2}) \big) \left(\frac{1}{\ln(k_1) - \ln(k_2)} - \frac{1}{\ln(k_1') - \ln(k_2')} \right) \right| \\
&\geq \frac{|\ln(w_{v_1}) - \ln(w_{v_2})|}{n^2}.
\end{aligned}
$$

Since w_{v_1} and w_{v_2} are independent, the maximum density of the right hand side is at most $\sigma^{-1} n^2$, and hence the probability that $|\rho - \rho'| \leq q$ is at most $2q \cdot \sigma^{-1} n^2$.

We now upper bound the number of tuple pairs that can appear in each case above. Each set $\tau(x_i)$ has at most n^4 elements, so there are at most $m^2 n^8$ pairs in Cases 1 and 2. There are at most n^4 choices of (k_1, k_2, k_1', k_2') for each (x, v_1, v_2) in Case 3, for a total of at most mn^6 pairs. The theorem now follows from the union bound. $\square$

Lastly, we formally state the existence of no-regret algorithms for the case of finite $|\mathcal{A}|$.

Fact 3.25 (E.g. [LW94])**.** *For a finite set of algorithms $\mathcal{A}$, there exists a randomized online learning algorithm L^* that, for every $m > 0$, has expected regret at most $O(\sqrt{(\log |\mathcal{A}|)/m})$ after seeing m instances. If the time cost of evaluating* $\textsc{cost}(A, x)$ *is bounded by B, then this algorithm runs in $O(B|\mathcal{A}|)$ time per instance.*

We can now state our main theorem.

Theorem 3.26 (Online Learning of Smooth MWIS)**.** *There is an online learning algorithm for σ-smooth MWIS that runs in time* $\mathrm{poly}(n, \sigma^{-1})$ *and has expected regret at most* $\mathrm{poly}(n^{-1})$ *(as $T \to \infty$).*

Proof. Fix a sufficiently large constant $d > 0$ and consider the first m instances of our sequence, $x_1, \ldots, x_m$, with $m = n^d \ln(\sigma^{-1})$. Let $q = 1/(n^d \cdot 4\sigma^{-1} m^2 n^8 \ln n)$. Let E_q be the event that every two distinct elements of $\tau(x_1) \cup \cdots \cup \tau(x_m)$ are at least q away from each other. By Lemma 3.24, E_q holds with probability at least $1 - 1/n^d$ over the randomness in the vertex weights.

Now, let $\mathcal{A}_N = \{A_i : i \in \{0, q, 2q, \ldots, \lfloor 1/q \rfloor q, 1\}\}$ be a "q-net." Our desired algorithm L is simply the algorithm L^* from Fact 3.25, applied to $\mathcal{A}_N$. We now analyze its expected regret.

If E_q does hold, then for every algorithm $A \in \mathcal{A}$, there is an algorithm $A' \in \mathcal{A}_N$ such that $\textsc{cost}(A, x_t) = \textsc{cost}(A', x_t)$ for $x_1, \ldots, x_m$. In other words, the best algorithm of $\mathcal{A}_N$ is no worse

than the best algorithm from all of $\mathcal{A}$, and in this case the expected regret of L is simply that of L^*. By Fact 3.25 and our choice of m, the expected regret (over the coin flips made by L^*) is at most inverse polynomial in n.

If E_q does not hold, our regret is at most 1, since COST is between 0 and 1. Averaging over the cases where E_q does and does not hold (with probabilities $1 - 1/n^d$ and $1/n^d$), the expected regret of the learning algorithm L (over the randomness in L^* and in the instances) is at most inverse polynomial in n. □

3.5 Conclusions and Future Directions

Empirical work on application-specific algorithm selection has far outpaced theoretical analysis of the problem, and this paper takes an initial step towards redressing this imbalance. We formulated the problem as one of learning the best algorithm or algorithm sequence from a class with respect to an unknown input distribution or input sequence. Many state-of-the-art empirical approaches to algorithm selection map naturally to instances of our learning frameworks. This chapter demonstrates that many well-studied classes of algorithms have small pseudo-dimension, and thus it is possible to learn a near-optimal algorithm from a relatively modest amount of data. While worst-case guarantees for no-regret online learning algorithms are impossible, good online learning algorithms exist in a natural smoothed model.

Our work suggests numerous wide-open research directions worthy of further study. For example:

1. Which computational problems admit a class of algorithms that simultaneously has low representation error and small pseudo-dimension (like in Section 3.3.4)?
2. Which algorithm classes can be learned online, in either a worst-case or a smoothed model?
3. When is it possible to learn a near-optimal algorithm using only a polynomial amount of computation, ideally with a learning algorithm that is better than brute-force search? Alternatively, are there (conditional) lower bounds stating that brute-force search is necessary for learning?[38]
4. Are there any non-trivial relationships between statistical learning measures of the complexity of an algorithm class and more traditional computational complexity measures?
5. How should instance features be chosen to minimize the representation error of the induced family of algorithm selection maps (cf., Section 3.3.5)?

[38] Recall the discussion in Section 3.2.3: even in practice, the state-of-the-art for application-specific algorithm selection often boils down to brute-force search.

Chapter 4

Learning Session Types from Traces

Chapter summary: We look at a specific problem in understanding how users are interacting with an application, given traces of their behavior. We model the problem as that of recovering a mixture of Markov chains, and show that under mild assumptions the mixture can be recovered from only its induced distribution on consecutive triples of states.

4.1 Introduction

The last scenario we look at is motivated by a specific problem in interaction design. Given an app or a website, we would like to understand how users are traveling through the app. The output of such an analysis might be a list of common flows, or a list of common but unexpected behaviors, due to e.g. users working around a user interface bug. We are not aware of any previous formulation of this task, either as an empirical or theoretical problem.

As a motivating example, consider the usage of a standard maps app on a phone. There are a number of different reasons one might use the app: to search for a nearby business, to get directions from one point to another, or just to orient oneself. Once the app has some users, we would like to algorithmically discover things like:

- Common uses for the app that the designers had not expected, or had not expected to be common. For instance, maybe a good fraction of users (or user sessions) simply use the app to check the traffic.
- Whether different types of users use the app differently. For instance, experienced users might use the app differently than first time users, either due to having different goals, or due to accomplishing the same tasks more efficiently.
- Undiscoverable flows, with users ignoring a simple, but hidden menu setting, and instead using a convoluted path to accomplish the same goal.
- Other unexpected behavior, such as users consistently pushing a button several times before the app responds, or doing other things to work around programming or UI errors.

In the previous two chapters, we imposed very weak restrictions on the instances we considered, and showed relatively strong results under those weak restrictions. We expect the most compelling results for this problem will also come from a "model-independent" point of view, à la Chapter 2, but initiate the study of the problem with a stronger set of assumptions.

We model the problem as follows. There are a small number L of unknown session types, such as "experienced user looking for food", or "first time user trying to figure out what the app does", or "user hitting bug #1234". There are $n \geq 2L$ observable states; the states can correspond to e.g. pages on a site, or actions (swipe, type, zoom) on an app. We assume user behavior is Markovian for each fixed session type; namely, each of the L session types corresponds to a Markov chain on the n states.

The app designers receive session trails of the following form: a session type and starting state is selected from some unknown distribution over $L \times n$, and then the session follows the type-specific Markov chain for some number of steps. The computational problem is to recover the transition probabilities in each of the L individual Markov chains, given a list of these trails. The L Markov chains can then be passed to a human for examination, past and future traces can be labelled with their most likely type, or users can be bucketed based on their typical session types.

Mathematically, this problem corresponds to learning a mixture of Markov chains, and is of independent interest. The rest of this chapter focuses on solving this problem.

4.1.1 Related Work

There are two immediate approaches to solving this problem. The first is to use the EM algorithm [DLR77]. The EM algorithm starts by guessing an initial set of parameters for the mixture, and then performs local improvements that increase the likelihood of the proposed solution. The EM algorithm is a useful benchmark and will converge to some local optimum, but it may be slow to get there [RW84], and there are no guarantees on the quality of the final solution.

The second approach is to model the problem as a Hidden Markov Model (HMM), and employ machinery for learning HMMs, particularly the recent tensor decomposition methods [AGH+14, AHK12, HKZ12]. As in our case, this machinery relies on having more observed states than hidden states. Unfortunately, directly modeling a Markov chain mixture as an HMM (or as a mixture of HMMs, as in [STS14]) requires nL hidden states for n observed states. Given that, one could try adapting the tensor decomposition arguments from [AHK12] to our problem, which is done in Section 4.3 of [Süb11]. However, as the author notes, this requires accurate estimates for the distribution of trajectories (or trails) of length five, whereas our results only require estimates for the distribution of trails of length three. This is a large difference in the amount of data one might need to collect, as one would expect to need $\Theta(n^t)$ samples to estimate the distribution of trails of length t.

An entirely different approach is to assume a Dirichlet prior on the mixture, and model the problem as learning a mixture of Dirichlet distributions [Süb11]. Besides requiring the Dirichlet prior, this method also requires very long trails (e.g., of minimum length 100).

A related line of work shows parameter recovery of various classes of models, using Kruskal's theorem on 3-way tables [AMR09]. Their models include HMMs as a special case. Dimension 3 seems to play a big role, as does generic identifiability (meaning, recovery of parameters for all but a measure zero subset of the possible parameter settings), both of which will show up here as well. Their results are not algorithmic in nature, but the connection is intriguing, and a potential avenue for future work.

4.1.2 Results

Let $\mathcal{S}$ be the distribution over session type and starting state, and let $\mathcal{M}$ be the transition probabilities for the L Markov chains. Let a t-trail be a trail of length t, namely, a starting state drawn from $\mathcal{S}$ followed by $t-1$ steps in the corresponding Markov chain in $\mathcal{M}$. Our main result is that under a weak non-degeneracy condition, $\mathcal{M}$ and $\mathcal{S}$ can be recovered exactly given their induced distribution over 3-trails.

4.2 Preliminaries

Let $[n] = \{1, \ldots, n\}$ be a state space. We consider Markov chains defined on $[n]$. For a Markov chain given by its $n \times n$ transition matrix M, let $M(i, j)$ denote the probability of moving from state i to state j. By definition, M is a stochastic matrix, $M(i, j) \geq 0$ and $\sum_j M(i, j) = 1$. (In general we use $A(i, j)$ to denote the (i, j)'th entry of a matrix A.)

For a matrix A, let $\overline{A}$ denote its transpose. Every $n \times n$ matrix A of rank r admits a singular value decomposition (SVD) of the form $A = U\Sigma\overline{V}$ where U and V are $n \times r$ orthogonal matrices and Σ is an $r \times r$ diagonal matrix with non-negative entries. For an $L \times n$ matrix B of full rank, its *right pseudoinverse* B^{-1} is an $n \times L$ matrix of full rank such that $BB^{-1} = I$; it is a standard fact that pseudoinverses exist and can be computed efficiently when $n \geq L$. Finally, for a subspace W, let $W^{\perp}$ denote its orthogonal subspace.

We now formally define a *mixture of Markov chains* $(\mathcal{M}, \mathcal{S})$. Let $L \geq 1$ be an integer. Let $\mathcal{M} = \{M^1, \ldots, M^L\}$ be L transition matrices, all defined on $[n]$. Let $\mathcal{S} = \{s^1, \ldots, s^L\}$ be a corresponding set of positive n-dimensional vectors of *starting probabilities* such that $\sum_{\ell,i} s_i^{\ell} = 1$. Given $\mathcal{M}$ and $\mathcal{S}$, a t-trail is generated as follows: first pick the chain ℓ and the starting state i with probability s_i^{ℓ}, and then perform a random walk according to the transition matrix M^{ℓ}, starting from i, for $t-1$ steps.

Throughout, we use i, j, k to denote states in $[n]$ and ℓ to denote a particular chain. Let 1_n be a column vector of n 1's.

Definition 4.1 (Reconstructing a Mixture of Markov Chains)**.** *Given a (large enough) set of trails generated by a mixture of Markov chains and an $L > 1$, find the parameters $\mathcal{M}$ and $\mathcal{S}$ of the mixture.*

Note that the number of parameters is $O(n^2 \cdot L)$. In this paper, we focus on a seemingly harder version of the reconstruction problem, where all of the given trails are of length three, i.e., every

trail is of the form $i \to j \to k$ for some three states $i, j, k \in [n]$. Surprisingly, we show that 3-trails are sufficient for perfect reconstruction.

By the definition of mixtures, the probability of generating a given 3-trail $i \to j \to k$ is

$$\sum_{\ell} s_i^{\ell} \cdot M^{\ell}(i, j) \cdot M^{\ell}(j, k), \tag{4.1}$$

which captures the stochastic process of choosing a particular chain ℓ using $\mathcal{S}$ and taking two steps in M^{ℓ}. Since we only observe the trails, the choice of the chain ℓ in the above process is latent. For each $j \in [n]$, let O_j be an $n \times n$ matrix such that $O_j(i, k)$ equals the value in (4.1). For simplicity, we assume we know each $O_j(i, k)$ exactly, rather than an approximation of it from samples.

We now give a simple decomposition of O_j in terms of the transition matrices in $\mathcal{M}$ and the starting probabilities in $\mathcal{S}$. Let P_j be the $L \times n$ matrix whose (ℓ, i)'th entry denotes the probability of using chain ℓ, starting in state i, and transitioning to state j, i.e., $P_j(\ell, i) = s_i^{\ell} \cdot M^{\ell}(i, j)$. In a similar manner, let Q_j be the $L \times n$ matrix whose (ℓ, k)'th entry denotes the probability of starting in state j, and transitioning to state k under chain ℓ, i.e., $Q_j(\ell, k) = s_j^{\ell} \cdot M^{\ell}(j, k)$. Finally, let $S_j = \text{diag}(s_j^1, \ldots, s_j^L)$ be the $L \times L$ diagonal matrix of starting probabilities in state j. Then,

$$O_j = \overline{P_j} \cdot S_j^{-1} \cdot Q_j. \tag{4.2}$$

This decomposition will form the key to our analysis.

4.3 Conditions for Unique Reconstruction

Before we delve into the details of the algorithm, we first identify a condition on the mixture $(\mathcal{M}, \mathcal{S})$ such that there is a unique solution to the reconstruction problem when we consider trails of length three. (To appreciate such a need, consider a mixture where two of the matrices M^{ℓ} and $M^{\ell'}$ in $\mathcal{M}$ are identical. Then for a fixed vector v, any s^{ℓ} and $s^{\ell'}$ with $s^{\ell} + s^{\ell'} = v$ will give the same observations, regardless of the length of the trails.) To motivate the condition we require, consider again the sets of $L \times n$ matrices $\mathcal{P} = \{P_1, \ldots, P_n\}$ and $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ as defined in (4.2). Together these matrices capture the $n^2L - 1$ parameters of the problem, namely, $n - 1$ for each of the n rows of each of the L transition matrices M^{ℓ}, and $nL - 1$ parameters defining $\mathcal{S}$. However, together $\mathcal{P}$ and $\mathcal{Q}$ have $2n^2L$ entries, implying algebraic dependencies between them.

Definition 4.2 (Shuffle pairs)**.** *Two ordered sets $\mathcal{X} = \{X_1, \ldots, X_n\}$ and $\mathcal{Y} = \{Y_1, \ldots, Y_n\}$ of $L \times n$ matrices are* shuffle pairs *if the j'th column of X_i is identical to the i'th column of Y_j for all $i, j \in [n]$.*

Note that $\mathcal{P}$ and $\mathcal{Q}$ are shuffle pairs. We state an equivalent way of specifying this definition. Consider a $2nL \times n^2$ matrix $\mathcal{A}(\mathcal{P}, \mathcal{Q})$ that consists of a top and a bottom half. The top half is an $nL \times n^2$ block diagonal matrix with P_i as the i'th block. The bottom half is a concatenation of n different $nL \times n$ block diagonal matrices; the i'th block of the j'th matrix is the j'th column of $-Q_i$. A representation of $\mathcal{A}$ is given in Figure 4.1. As intuition, note that in each column, the two blocks of L entries are the same up to negation.

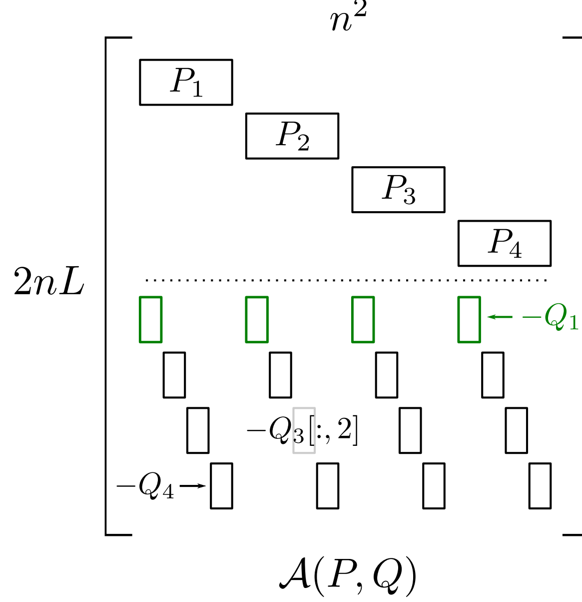

Figure 4.1: $\mathcal{A}(\mathcal{P}, \mathcal{Q})$ for $L = 2, n = 4$. When $\mathcal{P}$ and $\mathcal{Q}$ are shuffle pairs, each column has two copies of the same L-dimensional vector (up to negation). $\mathcal{M}$ is well-distributed if there are no non-trivial vectors v for which $v \cdot \mathcal{A}(\mathcal{P}, \mathcal{Q}) = 0$.

Let I_L be the $L \times L$ identity matrix, and let F be the $L \times 2nL$ matrix consisting of $2n$ $L \times L$ identity matrices in a row. It is straightforward to see that $\mathcal{P}$ and $\mathcal{Q}$ are shuffle pairs if and only if $F \cdot \mathcal{A}(\mathcal{P}, \mathcal{Q}) = 0$.

Let the *co-kernel* of a matrix X be the vector space comprising the vectors v for which $vX = 0$. We have the following definition.

Definition 4.3 (Well-distributed)**.** *The set of matrices $\mathcal{M}$ is* well-distributed *if the co-kernel of $\mathcal{A}(\mathcal{P}, \mathcal{Q})$ has rank L.*

Equivalently, $\mathcal{M}$ is well-distributed if the co-kernel of $\mathcal{A}(\mathcal{P}, \mathcal{Q})$ is spanned by the rows of F. Section 4.4 shows how to uniquely recover a mixture from the 3-trail probabilities O_j when $\mathcal{M}$ is well-distributed and $\mathcal{S}$ has only non-zero entries. In Section 4.5, we show that nearly all $\mathcal{M}$ are well-distributed, or more formally, that the set of non well-distributed $\mathcal{M}$ has (Lebesgue) measure 0.

4.4 A Reconstruction Algorithm

In this section we present an algorithm to recover a mixture from its induced distribution on 3-trails. We assume for the rest of the section that $\mathcal{M}$ is well-distributed (see Definition 4.3) and $\mathcal{S}$ has only non-zero entries, which also means P_j, Q_j, and O_j have rank L for each j.

At a high level, the algorithm begins by performing an SVD of each O_j, thus recovering both $\overline{P_j}$ and Q_j, as in (4.2), up to unknown rotation and scaling. The key to undoing the rotation will be

the fact that the matrices $\mathcal{P}$ and $\mathcal{Q}$ are shuffle pairs, and hence have algebraic dependencies.

More specifically, our algorithm consists of four high-level steps. We first list the steps and provide an informal overview; later we will describe each step in full detail.

(i) *Matrix decomposition*: Using SVD, we compute a decomposition $O_j = \overline{U_j}\Sigma_j V_j$ and let $P'_j = U_j$ and $Q'_j = \Sigma_j V_j$. These are the initial guesses at $(\overline{P_j}, Q_j)$. We prove in Lemma 4.4 that there exist $L \times L$ matrices Y_j and Z_j so that $P_j = Y_j P'_j$ and $Q_j = Z_j Q'_j$ for each $j \in [n]$.

(ii) *Co-kernel*: Let $\mathcal{P}' = \{P'_1, \ldots, P'_n\}$, and $\mathcal{Q}' = \{Q'_1, \ldots, Q'_n\}$. We compute the co-kernel of matrix $\mathcal{A}(\mathcal{P}', \mathcal{Q}')$ as defined in Section 4.3, to obtain matrices Y'_j and Z'_j. We prove that there is a single matrix R for which $Y_j = RY'_j$ and $Z_j = RZ'_j$ for all j.

(iii) *Diagonalization*: Let $\overline{R'}$ be the matrix of eigenvectors of $(Z'_1\overline{Y'_1})^{-1}(Z'_2\overline{Y'_2})$. We prove that there is a permutation matrix Π and a diagonal matrix D such that $R = D\Pi R'$.

(iv) *Two-trail matching*: Given O_j it is easy to compute the probability distribution of the mixture over 2-trails. We use these to solve for D, and using D, compute R, Y_j, P_j, and S_j for each j. ordering of $[L]$).

4.4.1 Matrix decomposition

From the definition, both P'_j and Q'_j are $L \times n$ matrices of full rank. The following lemma states that the SVD of the product of two matrices $\overline{A}$ and B returns the original matrices up to a change of basis.

Lemma 4.4. *Let A, B, C, D be $L \times n$ matrices of full rank, such that $\overline{A}B = \overline{C}D$. Then there is an $L \times L$ matrix X of full rank such that $C = X^{-1}A$ and $D = \overline{X}B$.*

Proof. Note that $\overline{A} = \overline{A}BB^{-1} = \overline{C}DB^{-1} = \overline{C}W$ for $W = DB^{-1}$. Since A has full rank, W must as well. We then get $\overline{C}D = \overline{A}B = \overline{C}WB$, and since $\overline{C}$ has full column rank, $D = WB$. Setting $X = \overline{W}$ completes the proof. □

Since $O_j = \overline{P_j}(S_j^{-1}Q_j)$ and $O_j = \overline{P'_j}Q'_j$, Lemma 4.4 implies that there exists an $L \times L$ matrix X_j of full rank such that $P_j = X_j^{-1}P'_j$ and $Q_j = S_j\overline{X_j}Q'_j$. Let $Y_j = X_j^{-1}$, and let $Z_j = S_j\overline{X_j}$. Note that both Y_j and Z_j have full rank, for each j. Once we have Y_j and Z_j, we can easily compute both P_j and S_j, so we have reduced our problem to finding Y_j and Z_j.

4.4.2 Co-kernel

Since $(\mathcal{P}, \mathcal{Q})$ is a shuffle pair, $((Y_j P'_j)_{j\in[n]}, (Z_j Q'_j)_{j\in[n]})$ is also a shuffle pair. We can write the latter fact as $\mathcal{B}(Y, Z)\,\mathcal{A}(P', Q') = 0$, where $\mathcal{B}(Y, Z)$ is the $L \times 2nL$ matrix comprising $2n$ matrices concatenated together; first Y_j for each j, and then Z_j for each j (see Figure 4.2). We know $\mathcal{A}(P', Q')$ from the matrix decomposition step, and we are trying to find $\mathcal{B}(Y, Z)$.

$$L \left[\boxed{Y_1}\boxed{Y_2}\boxed{Y_3}\boxed{Y_4}\boxed{Z_1}\boxed{Z_2}\boxed{Z_3}\boxed{Z_4} \right]$$

$2nL$ (width); $\mathcal{B}(Y, Z)$

Figure 4.2: $\mathcal{B}(Y, Z)$ for $L = 2, n = 4$. $\mathcal{B}(Y, Z)$ is the co-kernel of $\mathcal{A}(\mathcal{P}', \mathcal{Q}')$.

By well-distributedness, the co-kernel of $\mathcal{A}(P, Q)$ has rank L. Let D be the $2nL \times 2nL$ block diagonal matrix with the diagonal entries $(Y_1^{-1}, Y_2^{-1}, \ldots, Y_n^{-1}, Z_1^{-1}, Z_2^{-1}, \ldots, Z_n^{-1})$. Then $\mathcal{A}(P', Q') = D\,\mathcal{A}(P, Q)$. Since D has full rank, the co-kernel of $\mathcal{A}(P', Q')$ has rank L as well.

We compute an arbitrary basis of the co-kernel of $\mathcal{A}(P', Q')$),[1] and write it as an $L \times 2nL$ matrix as an initial guess $\mathcal{B}(Y', Z')$ for $\mathcal{B}(Y, Z)$. Since $\mathcal{B}(Y, Z)$ lies in the co-kernel of $\mathcal{A}(P', Q')$, and has exactly L rows, there exists an $L \times L$ matrix R such that $B(Y, Z) = R\,B(Y', Z')$, or equivalently, such that $Y_j = RY_j'$ and $Z_j = RZ_j'$ for every j. Since Y_j and Z_j have full rank, so does R. Now our problem is reduced to computing R.

4.4.3 Diagonalization

Recall from the matrix decomposition step that there exist matrices X_j such that $Y_j = X_j^{-1}$ and $Z_j = S_j\overline{X_j}$. Hence $Z_j'\overline{Y_j'} = (R^{-1}Z_j)(\overline{Y_j}\,\overline{R^{-1}}) = R^{-1}S_j\overline{R^{-1}}$. It seems difficult to compute R directly from equations of the form $R^{-1}S_j\overline{R^{-1}}$, but we can multiply any two of them together to get, e.g., $(Z_1'\overline{Y_1'})^{-1}(Z_2'\overline{Y_2'}) = \overline{R}S_1^{-1}S_2\overline{R^{-1}}$.

Since $S_1^{-1}S_2$ is a diagonal matrix, we can diagonalize $\overline{R}S_1^{-1}S_2\overline{R^{-1}}$ as a step towards computing R. Let $\overline{R'}$ be the matrix of eigenvectors of $\overline{R}S_1^{-1}S_2\overline{R^{-1}}$. Now, $\overline{R}$ is determined up to a scaling and ordering of the eigenvectors. In other words, there is a permutation matrix Π and diagonal matrix D such that $R = D\Pi R'$.

4.4.4 Two-Trail Matching

First, $O_j 1_n = \overline{P_j}S_j^{-1}Q_j 1_n = \overline{P_j}1_L$ for each j, since each row of $S_j^{-1}Q_j$ is simply the set of transition probabilities out of a particular Markov chain and state. Another way to see it is that both $O_j 1_n$ and $\overline{P_j}1_L$ are vectors whose i'th coordinate is the probability of the trail $i \to j$.

From the first three steps of the algorithm, we also have $P_j = Y_j P_j' = RY_j'P_j' = D\Pi R'Y_j'P_j'$.

Hence $\overline{1_L}D\Pi = \overline{1_L}P_1(R'Y_1'P_1')^{-1} = \overline{O_1 1_n}(R'Y_1'P_1')^{-1}$, where the inverse is a pseudoinverse. We arbitrarily fix Π, from which we can compute D, R, Y_j, and finally P_j for each j. From the diagonalization step (Section 4.4.3), we can also compute $S_j = R(Z_j'\overline{Y_j'})\overline{R}$ for each j.

Note that the algorithm implicitly includes a proof of uniqueness, up to a setting of Π. Different orderings of Π correspond to different orderings of M^ℓ in M.

[1] For instance, by taking the SVD of $\mathcal{A}(P', Q')$, and looking at the singular vectors.

4.4.5 Modifications for Noisy Data

The algorithm is easy to implement, and so we implemented and tested it on synthetic data. We generate mixtures with each entry of $\mathcal{M}$ and $\mathcal{S}$ chosen uniformly at random (and then appropriately normalized). For each mixture, we take a finite number of samples, and use that to construct an empirical O_j. We then see if our algorithm recovers something close to $\mathcal{M}$ and $\mathcal{S}$. In the process, we observed three modifications that seemed to make the algorithm more robust in this somewhat noisy setting.

First, the matrices P_j that we recover at the end need not be stochastic, or even real-valued. Following the work of [CSC+13], we first taking the norm of every entry, and then normalize so that each of the columns sums to 1.

Next, in the diagonalization step (Section 4.4.3), we sum $(Z_i'\overline{Y_i})^{-1}(Z_{i+1}'\overline{Y_{i+1}})^{-1}$ over all i before diagonalizing to estimate R', instead of just using $i = 1$. This gives some improvement, though we suspect there is likely a much better/more sophisticated way to combine the information in the n matrices.

Finally, given $\mathcal{M}$, the observations O_j are a linear function of $\mathcal{S}$. Rather than setting $S_j = R(Z_j'\overline{Y_j'})\overline{R}$ as in Section 4.4.4, we can compute the $\mathcal{S}$ that minimizes the L_1 distance between the reconstructed O_j and the empirical O_j via a straightforward linear program.

4.5 Analysis

We now show that nearly all $\mathcal{M}$ are well-distributed (see Definition 4.3), or more formally, that the set of non well-distributed $\mathcal{M}$ has (Lebesgue) measure 0 for every $L > 1$ and $n \geq 2L$.

We first introduce some notation. All arrays and indices are 1-indexed. In previous sections, we have interpreted i, j, k, and ℓ as states or as indices of a mixture; in this section we drop those interpretations and just use them as indexes.

For vectors $v_1, \ldots, v_n \in \mathbb{R}^L$, let $v_{[n]}$ denote $(v_1, \ldots, v_n)$, and let $*(v_1, \ldots, v_n)$ denote the v_i's concatenated together to form a vector in $\mathbb{R}^{nL}$. Let $v_i[j]$ denote the j'th coordinate of vector v_i.

We first show that there exists at least one well-distributed $\mathcal{P}$ for each n and L.

Lemma 4.5 (Existence of a well-distributed $\mathcal{P}$)**.** *For every n and L with $n \geq 2L$, there exists a $\mathcal{P}$ for which the co-kernel of $\mathcal{A}(\mathcal{P}, \mathcal{Q})$ has rank L.*

Proof. It is sufficient to show it for $n = 2L$, since for larger n we can pad with zeros. Also, recall that $F \cdot \mathcal{A}(\mathcal{P}, \mathcal{Q}) = 0$ for any $\mathcal{P}$, where F is the $L \times 2nL$ matrix consisting of $2n$ identity matrices concatenated together. So the co-kernel of any $\mathcal{A}(\mathcal{P}, \mathcal{Q})$ has rank at least L, and we just need to show that there exists a $\mathcal{P}$ where the co-kernel of $\mathcal{A}(\mathcal{P}, \mathcal{Q})$ has rank at most L.

Now, let e_l be the l'th basis vector in $\mathbb{R}^L$. Let $\mathcal{P}^* = (P_1^*, \ldots, P_n^*)$, and let p_{ij}^* denote the j'th column of P_i^*. We set p_{ij}^* to the (i, j)'th entry of

$$\begin{pmatrix} e_1 & e_2 & \cdots & e_L & e_1 & e_2 & \cdots & e_L \\ e_L & e_1 & \cdots & e_{L-1} & e_L & e_1 & \cdots & e_{L-1} \\ \vdots & & & \vdots & \vdots & & & \vdots \\ e_2 & e_3 & \cdots & e_1 & e_2 & e_3 & \cdots & e_1 \\ e_1 & e_2 & \cdots & e_L & e_L & e_1 & \cdots & e_{L-1} \\ e_L & e_1 & \cdots & e_{L-1} & e_{L-1} & e_L & \cdots & e_{L-2} \\ \vdots & & & \vdots & \vdots & & & \vdots \\ e_2 & e_3 & \cdots & e_1 & e_1 & e_2 & \cdots & e_L \end{pmatrix}.$$

Formally, $p^*_{ij} = \begin{cases} e_{j-i-1} & \text{if } i \leq L \text{ or } j \leq L \\ e_{j-i}, & \text{if } i, j > L \end{cases}$, where subscripts are taken mod L. For intuition, note that we can split the above matrix into four $L \times L$ blocks $\begin{pmatrix} E & E \\ E & E' \end{pmatrix}$ where E' is a horizontal "rotation" of E.

Now, let $a_{[n]}, b_{[n]}$ be any vectors in $\mathbb{R}^L$ such that $v = *(a_1, \ldots, a_n, b_1, \ldots, b_n) \in \mathbb{R}^{2nL}$ is in the co-kernel of $\mathcal{A}(\mathcal{P}^*, \mathcal{Q}^*)$. Recall this means $v \cdot \mathcal{A}(\mathcal{P}^*, \mathcal{Q}^*) = 0$. Writing out the matrix $\mathcal{A}$, it is not too hard to see that this holds if and only if

$$\langle a_i, p^*_{ij} \rangle = \langle b_j, p^*_{ij} \rangle$$

for each i and j.

Consider the i and j where $p^*_{ij} = e_1$. For each $k \in [L]$, we have $a_k[1] = b_k[1]$ from the upper left quadrant, $a_k[1] = b_{L+k}[1]$ from the upper right quadrant, $a_{L+k}[1] = b_k[1]$ from the lower left quadrant, and $a_{L+k}[1] = b_{L+(k+1 \pmod L)}[1]$ from the lower right quadrant. It is easy to see that these combine to imply that $a_i[1] = b_j[1]$ for all $i, j \in [n]$.

A similar argument for each $l \in [L]$ shows that $a_i[l] = b_j[l]$ for all i, j and l. Equivalently, $a_i = b_j$ for each i and j, which means that v lives in a subspace of dimension L, as desired. □

We now bootstrap from our one example to show that almost all $\mathcal{P}$ are well-distributed.

Theorem 4.6 (Almost all $\mathcal{P}$ are well-distributed)**.** *The set of non-well-distributed $\mathcal{P}$ has Lebesgue measure 0 for every n and L with $n \geq 2L$.*

Proof. Let $\mathcal{A}'(\mathcal{P}, \mathcal{Q})$ be all but the last L rows of $\mathcal{A}(\mathcal{P}, \mathcal{Q})$. For any $\mathcal{P}$, let

$$h(\mathcal{P}) = \det |\mathcal{A}'(\mathcal{P}, \mathcal{Q})\, \overline{\mathcal{A}'(\mathcal{P}, \mathcal{Q})}|.$$

Note that $h(\mathcal{P})$ is non-zero if and only if $\mathcal{P}$ is well-distributed. Let $\mathcal{P}^*$ be the $\mathcal{P}^*$ from Lemma 4.5. Since $\mathcal{A}'(\mathcal{P}^*, \mathcal{Q}^*)$ has full row rank, $h(\mathcal{P}^*) \neq 0$. Since h is a polynomial function of the entries of $\mathcal{P}$, and h is non-zero somewhere, h is non-zero almost everywhere [CT05]. □

The above theorem is tight in the sense that no $\mathcal{P}$ can be well-distributed if $n < 2L$.

Bibliography

[AB99] M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations.* Cambridge University Press, 1999. 44, 48, 58, 59, 63

[AC09] Reid Andersen and Kumar Chellapilla. Finding dense subgraphs with size bounds. In *Algorithms and Models for the Web-Graph*, pages 25–37. Springer, 2009. 31

[ACCL06] N. Ailon, B. Chazelle, S. Comandur, and D. Liu. Self-improving algorithms. In *Proceedings of the Symposium on Discrete Algorithms*, pages 261–270, 2006. 44, 45, 46, 55, 56, 57

[ADH+08] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):241–249, 2008. 29, 31

[AGH+14] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *Journal of Machine Learning Research*, 15(1):2773–2832, 2014. 72

[AGK+04] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. Local search heuristics for k-median and facility location problems. *SIAM Journal on Computing*, 33(3):544–562, 2004. 54

[AHK12] Animashree Anandkumar, Daniel Hsu, and Sham M Kakade. A method of moments for mixture models and hidden Markov models. In *Proceedings of the Conference on Learning Theory*, pages 33.1–33.34, 2012. 72

[AJB00] R. Albert, H. Jeong, and A.-L. Barabási. Error and attack tolerance of complex networks. *Nature*, 406:378–382, 2000. 9

[AMR09] Elizabeth S Allman, Catherine Matias, and John A Rhodes. Identifiability of parameters in latent structure models with many observed variables. *The Annals of Statistics*, pages 3099–3132, 2009. 73

[BA99] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999. 8

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(1):281–305, 2012. 46

[BBG13] M.-F. Balcan, A. Blum, and A. Gupta. Clustering under approximation stability. *Journal of the ACM*, 60(2):1068–1077, 2013. 12, 26, 27

[BDGL08] I. Bordino, D. Donata, A. Gionis, and S. Leonardi. Mining large networks with subgraph counting. In *Proceedings of the International Conference on Data Mining*, pages 737–742, 2008. 31

[BKM+00] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks*, 33:309–320, 2000. 8

[BNR03] Allan Borodin, Morten N. Nielsen, and Charles Rackoff. (Incremental) priority algorithms. *Algorithmica*, 37(4):295–326, 2003. 50

[Bur04] R. S. Burt. Structural holes and good ideas. *American Journal of Sociology*, 110(2):349–399, 2004. 13

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, 2004. 60

[CBO14] U. S. Congressional Budget Office: The budget and economic outlook: 2015 to 2025. 2014. 45

[CC11] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proceedings of Knowledge, Discovery, and Data Mining*, pages 672–680, 2011. 29

[CF06] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys*, 38(1), 2006. 8, 13

[CJ12] A. Chandrasekhar and M. Jackson. Tractable and consistent random graph models. Technical Report 1210.7375, Arxiv, 2012. 29

[CL02a] F. Chung and L. Lu. The average distances in random graphs with given expected degrees. *Proceedings of the National Academy of Sciences*, 99(25):15879–15882, 2002. 8, 13

[CL02b] F. Chung and L. Lu. Connected components in random graphs with given degree sequences. *Annals of Combinatorics*, 6:125–145, 2002. 8

[CL06] Nicolo Cesa-Bianchi and Gábor Lugosi. *Prediction, learning, and games*. Cambridge University Press, 2006. 64, 65

[CMS10] Kenneth L. Clarkson, Wolfgang Mulzer, and C. Seshadhri. Self-improving algorithms for convex hulls. In *Proceedings of the Symposium on Discrete Algorithms*, pages 1546–1565, 2010. 45, 55

[CMS12] Kenneth L. Clarkson, Wolfgang Mulzer, and C. Seshadhri. Self-improving algorithms for coordinate-wise maxima. In *Proceedings of the Symposium on Computational Geometry*, pages 277–286, 2012. 45, 55

[Coh09] Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engineering*, 11:29–41, 2009. 30

[Col88] J. S. Coleman. Social capital in the creation of human capital. *American Journal of Sociology*, 94:95–120, 1988. 13

[CS08] K. L. Clarkson and C. Seshadhri. Self-improving algorithms for Delaunay triangulations. In *Proceedings of the Symposium on Computational Geometry*, pages 148–155, 2008. 45, 55

[CSC+13] S. B. Cohen, K. Stratos, M. Collins, D. P. Foster, and L. Ungar. Experiments with spectral learning of latent-variable PCFGs. In *Proceedings of the North American Chapter of the Association for Computational Linguistics*, 2013. 78

[CT05] Richard Caron and Tim Traynor. The zero set of a polynomial. `http://www1.uwindsor.ca/math/sites/uwindsor.ca.math/files/05-03.pdf`, 2005. 79

[CZF04] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the Conference on Data Mining*, pages 442–446, 2004. 8

[Dev86] Luc Devroye. *Lectures Notes on Bucket Algorithms*. Birkhäuser, 1986. 55

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977. 72

[Fau06] Katherine Faust. Comparing social networks: Size, density, and local structure. *Metodoloski Zvezki*, 3(2):185–216, 2006. 13

[FFF99] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of SIGCOMM*, pages 251–262, 1999. 8

[Fin98] Eugene Fink. How to solve it automatically: Selection among problem solving methods. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 128–136, 1998. 44

[For10] S. Fortunato. Community detection in graphs. *Physics Reports*, 486:75–174, 2010. 8

[FPP06] A. Ferrante, G. Pandurangan, and K. Park. On the hardness of optimization in power law graphs. In *Proceedings of the Conference on Computing and Combinatorics*, pages 417–427, 2006. 13

[FWVDC10] B. Foucault Welles, A. Van Devender, and N. Contractor. Is a friend a friend?: Investigating the structure of friendship networks in virtual worlds. In *Extended Abstracts on Human Factors in Computing Systems*, pages 4027–4032, 2010. 13

[Gat14] Andreas Gathmann. Lectures notes on algebraic geometry. 2014. 54

[GK07] J. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Research in Computational Molecular Biology*, pages 92–106, 2007. 29, 31

[GKV16] R. Gupta, R. Kumar, and S. Vassilvitskii. On mixtures of markov chains. In *Proceedings of Advances in Neural Information Processing Systems*, 2016. 7

[GN02] M. Girvan and M. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002. 8

[GR16] Rishi Gupta and Tim Roughgarden. A PAC approach to application-specific algorithm selection. In *Proceedings of Innovations in Theoretical Computer Science*, 2016. 7

[GRS14] Rishi Gupta, Tim Roughgarden, and C Seshadhri. Decompositions of triangle-dense graphs. In *Proceedings of Innovations in Theoretical Computer Science*, pages 471–482. ACM, 2014. 7

[GRS16] R. Gupta, T. Roughgarden, and C. Seshadhri. Decompositions of triangle-dense graphs. 2016. 7

[Has96] J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of Foundations of Computer Science*, pages 627–636, 1996. 31

[Hau92] D. Haussler. Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Information and Computation*, 100(1):78–150, 1992. 44

[HBPS07] F. Hormozdiari, P. Berenbrink, N. Prulj, and S. Cenk Sahinalp. Not all scale-free networks are born equal: The role of the seed graph in PPI network evolution. *PLoS Computational Biology*, 118, 2007. 29, 31

[HJY+10] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Proceedings of Advances in Neural Information Processing Systems*, pages 883–891, 2010. 44

[HKZ12] Daniel Hsu, Sham M Kakade, and Tong Zhang. A spectral algorithm for learning hidden Markov models. *Journal of Computer and System Sciences*, 78(5):1460–1480, 2012. 72

[HL70] P. W. Holland and S. Leinhardt. A method for detecting structure in sociometric data. *American Journal of Sociology*, 76:492–513, 1970. 13

[HRG+01] Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Henry A. Kautz, Bart Selman, and David Maxwell Chickering. A Bayesian approach to tackling hard computational problems. In *Proceedings of the Conference in Uncertainty in Artificial Intelligence*, pages 235–244, 2001. 44

[HXHL14] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014. 44, 59

[IFMN12] Joseph J. Pfeiffer III, Timothy La Fond, Sebastián Moreno, and Jennifer Neville. Fast generation of large scale social networks while incorporating transitive closures. In

Proceedings of the International Conference on Privacy, Security, Risk and Trust, pages 154–165, 2012. 13

[IWM00] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of Pacific-Asia KDD*, pages 13–23, 2000. 29

[Jam06] G. J. O. Jameson. Counting zeros of generalized polynomials: Descartes rule of signs and Laguerres extensions. *Mathematical Gazette*, 90(518):223–234, 2006. 55

[JM97] David S Johnson and Lyle A McGeoch. The traveling salesman problem: A case study in local optimization. volume 1, pages 215–310. 1997. 54

[KGM12] Lars Kotthoff, Ian P. Gent, and Ian Miguel. An evaluation of machine learning in algorithm selection for search problems. *AI Communications*, 25(3):257–270, 2012. 44

[KK04] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9), 2004. 29

[Kle00a] J. M. Kleinberg. Navigation in a small world. *Nature*, 406:845, 2000. 8

[Kle00b] J. M. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the Symposium on Theory of Computing*, pages 163–170, 2000. 8

[Kle02] J. M. Kleinberg. Small-world phenomena and the dynamics of information. In *Proceedings of Advances in Neural Information Processing Systems*, volume 1, pages 431–438, 2002. 8

[Knu75] D. E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975. 47, 57

[KP06] S. Khot and A. Ponnuswami. Better inapproximability results for max-clique, chromatic number and min-3lin-deletion. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pages 226–237, 2006. 31

[KRR+00] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic models for the web graph. In *Proceedings of Foundations of Computer Science*, pages 57–65, 2000. 8

[LAS+08] H. Lin, C. Amanatidis, M. Sideri, R. M. Karp, and C. H. Papadimitriou. Linked decompositions of networks and the power of choice in Polya urns. In *Proceedings of the Symposium on Discrete Algorithms*, pages 993–1002, 2008. 9

[LCK+10] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11:985–1042, 2010. 8

[LKF07] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1):article no. 2, March 2007. 13

[LLDM08] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2008. 8

[LNS09] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4), 2009. 44, 58

[Lon01] Philip M. Long. Using the pseudo-dimension to analyze approximation algorithms for integer programming. In *Proceedings of the International Workshop on Algorithms and Data Structures*, pages 26–37, 2001. 44

[LOS02] Daniel Lehmann, Liadan Ita O'Callaghan, and Yoav Shoham. Truth revelation in approximately efficient combinatorial auctions. *Journal of the ACM*, 49(5):577–602, 2002. 45

[LT79] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. 13

[LW94] Nick Littlestone and Manfred K Warmuth. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994. 69

[MM14] Mehryar Mohri and Andres Munoz Medina. Learning theory and algorithms for revenue optimization in second price auctions with reserve. In *Proceedings of the International Conference on Machine Learning*, pages 262–270, 2014. 44

[MR15] Jamie H Morgenstern and Tim Roughgarden. The pseudo-dimension of near-optimal auctions. In *Proceedings of Advances in Neural Information Processing Systems*, pages 136–144, 2015. 44

[MS10] A. Montanari and A. Saberi. The spread of innovations in social networks. *Proceedings of the National Academy of Sciences*, 107(47):20196–20201, 2010. 9

[MS14] Paul Milgrom and Ilya Segal. Deferred-acceptance auctions and radio spectrum reallocation. Working paper, 2014. 45

[New01] M. E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, 98(2):404–409, 2001. 8

[New03] M. E. J. Newman. Properties of highly clustered networks. *Physical Review E*, 68(2):026121, 2003. 8

[New06] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(3):036104, 2006. 8

[NS85] J. Nesetril and P. Svatopluk. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985. 30

[PCJ04] N. Przulj, D. G. Corneil, and I. Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004. 29, 31

[PSK12] Ali Pinar, C. Seshadhri, and Tamara G. Kolda. The similarity between Stochastic Kronecker and Chung-Lu graph models. In *Proceedings of the SIAM Conference on Data Mining*, 2012. 13

[RBH12] M. Rahman, M. Bhuiyan, and M. Al Hasan. Graft: an approximate graphlet counting algorithm for large graph analysis. In *Proceedings of the Conference on Information and Knowledge Management*, pages 1467–1471, 2012. 29, 31

[RGGP13] R. Rossi, D. Gleich, A. Gebremedhin, and Md. Patwary. Parallel maximum clique algorithms with applications to network analysis and storage. Technical Report 1302.6256, Arxiv, 2013. 31

[Rou14] Tim Roughgarden. Lecture notes on beyond worst-case analysis, December 2014. 1

[RS86] N. Robertson and P. D. Seymour. Graph minors III: Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1986. 13

[RW84] R. A. Redner and H. F. Walker. Mixture densities, maximum likelihood, and the EM algorithm. *SIAM Review*, 26:195–239, 1984. 72

[SBD06] Nathan Srebro and Shai Ben-David. Learning bounds for support vector machines with learned kernels. In *Proceedings of the 19th Annual Conference on Learning Theory*, pages 169–183, 2006. 44

[SCW+10] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao. Measurement-calibrated graph models for social network experiments. In *Proceedings of the World Wide Web Conference*, pages 861–870. ACM, 2010. 8, 13

[SKP12] C. Seshadhri, Tamara G. Kolda, and Ali Pinar. Community structure and scale-free collections of Erdös-Rényi graphs. *Physical Review E*, 85(5):056109, May 2012. 13

[SNA] SNAP. Stanford Network Analysis Project (SNAP). Available at `http://snap.stanford.edu/`. 12, 30, 32

[SPK13] C. Seshadhri, A. Pinar, and T. G. Kolda. Fast triangle counting through wedge sampling. In *Proceedings of the SIAM Conference on Data Mining*, 2013. 13, 29, 30, 31

[SPR11] V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. In *Proceedings of ACM SIGMOD*, pages 721–732, 2011. 14, 15

[ST09] Daniel A Spielman and Shang-Hua Teng. Smoothed analysis: an attempt to explain the behavior of algorithms in practice. *Communications of the ACM*, 52(10):76–84, 2009. 5, 67

[STS14] Cem Subakan, Johannes Traa, and Paris Smaragdis. Spectral learning of mixture of hidden Markov models. In *Proceedings of Advances in Neural Information Processing Systems*, pages 2249–2257, 2014. 72

[STY03] Shuichi Sakai, Mitsunori Togasaki, and Koichi Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Applied Mathematics*, 126(2):313–322, 2003. 45

[Süb11] Yusuf Cem Sübakan. Probabilistic time series classification. Master's thesis, Boğaziçi University, 2011. 72

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the World Wide Web Conference*, pages 607–614, 2011. 30

[SW05] Thomas Schank and Dorothea Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9:265–275, 2005. 30

[Sze78] E. Szemerédi. Regular partitions of graphs. *Problèmes combinatoires et théorie des graphes*, 260:399–401, 1978. 14

[TBG+13] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of Knowledge Discovery and Data Mining*, pages 104–112, 2013. 31

[TKMF09] C. Tsourakakis, U. Kang, G. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of Knowledge Discovery and Data Mining*, pages 837–846, 2009. 29, 30, 31

[Tso08] Charalampos E. Tsourakakis. Fast counting of triangles in large real networks, without counting: Algorithms and laws. In *Proceedings of the International Conference on Data Mining*, pages 608–617, 2008. 29, 30

[UBK13] J. Ugander, L. Backstrom, and J. Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the World Wide Web Conference*, pages 1307–1318, 2013. 13, 29, 38

[UKBM11] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the Facebook social graph. Technical Report 1111.4503, Arxiv, 2011. 3, 8, 9, 13

[VB12] J. Vivar and D. Banks. Models for networks: a cross-disciplinary science. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(1):13–27, 2012. 13

[WF94] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994. 9

[WS98] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998. 8, 13

[XHHL08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. 47, 57, 58

[XHHL11] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *Proceedings of the RCRA workshop on combinatorial explosion at the International Joint Conference on Artificial Intelligence*, pages 16–30, 2011. 46

[XHHL12] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2012. 47

[YH02] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the International Conference on Data Mining*, pages 721–724, 2002. 29