12-18-2020

ALGORITHMS FOR MASSIVE, EXPENSIVE, OR OTHERWISE INCONVENIENT GRAPHS

David Tench

ALGORITHMS FOR MASSIVE, EXPENSIVE, OR OTHERWISE INCONVENIENT GRAPHS

A Dissertation Presented

by

DAVID TENCH

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2020

College of Information and Computer Sciences

ALGORITHMS FOR MASSIVE, EXPENSIVE, OR OTHERWISE INCONVENIENT GRAPHS

A Dissertation Presented

by

DAVID TENCH

Approved as to style and content by:

_____

Andrew McGregor, Chair

_____

Phillipa Gill, Member

_____

Markos Katsoulakis, Member

_____

Cameron Musco, Member

_____

James Allan, Chair of the Faculty
College of Information and Computer Sciences

ABSTRACT

ALGORITHMS FOR MASSIVE, EXPENSIVE, OR OTHERWISE INCONVENIENT GRAPHS

SEPTEMBER 2020

DAVID TENCH

B.Sc., LEHIGH UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew McGregor

A long-standing assumption common in algorithm design is that any part of the input is accessible at any time for unit cost. However, as we work with increasingly large data sets, or as we build smaller devices, we must revisit this assumption. In this thesis, I present some of my work on graph algorithms designed for circumstances where traditional assumptions about inputs do not apply.

1. Classical graph algorithms require direct access to the input graph and this is not feasible when the graph is too large to fit in memory. For computation on massive graphs we consider the dynamic streaming graph model. Given an input graph defined by as a stream of edge insertions and deletions, our goal is to approximate properties of this graph using space that is sublinear in the size of the stream. In this thesis, I present algorithms for approximating vertex connectivity, hypergraph edge connectivity, maximum coverage, unique coverage, and temporal connectivity in graph streams.

2. In certain applications the input graph is not explicitly represented, but its edges may be discovered via queries which require costly computation or measurement. I present two open-source systems which solve real-world problems via graph algorithms which may access their inputs only through costly edge queries. MESH is a memory manager which compacts memory efficiently by finding an approximate graph matching subject to stringent time and edge query restrictions. PathCache is an efficiently scalable network measurement platform that outperforms the current state of the art.

CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

When designing and analyzing algorithms it is typically assumed that the input is easily accessible. For example, when designing an algorithm to sort an array of integers we assume that we can write to or read from any position in the array at any moment, and each such operation can be performed very quickly. In such a case we say that we have *random access* to the input, meaning that any part of the input may be accessed at any time for unit cost.

The vast growth in recent years of the scope of computing and data science challenges often leads to circumstances which complicate this standard model of computation. For instance, the input we wish to run an algorithm on may be massive – larger than can fit in the RAM of available computers. An input may be distributed across many different storage devices, or accessible only via noisy or expensive sensors. Such conditions have the potential to violate the random access assumption: perhaps we are only able to access some subset of the input at any time, or maybe we pay a significant cost for any access operation.

In this thesis we investigate algorithmic challenges in two broad settings where aspects of the random access assumption fail: the *streaming* domain, where a massive input is only accessible as an arbitrarily-ordered sequence of elements and working memory is sharply limited; and *query-accessible* inputs, for which accessing a piece of the input requires the algorithm to pay a high price in computation time, energy, money, durability, or some other scarce resource.

1.1 The Graph Streaming Setting

Massive graphs arise in many applications. Popular examples include the web-graph, social networks, and biological networks but, more generally, graphs are a natural abstraction whenever we have information about both a set of basic entities and relationships between these entities. Unfortunately, it is not possible to use existing algorithms to process many of these graphs; many of these graphs are too large to be stored in main memory and are constantly changing. Rather, there is a growing need to design new algorithms for even basic graph problems in the relevant computational models.

In Chapters 2, 3, and 4, we consider algorithms in the data stream and linear sketching models. In the data stream model, a sequence of edge insertions (and possibly deletions) defines an input graph and the goal is to solve a specific problem on this graph given only one-way access to the input sequence and limited working memory. While insert-only graph streaming has been an active area of research for almost a decade, it is only relatively recently algorithms have been found that handle insertions and deletions [6–8, 75, 93, 94, 107]. We refer to streams with insertions and deletions as *dynamic* graph streams. The main technique

used in these algorithms is *linear sketching* where a random linear projection of the input graph is maintained as the graph is updated. To be useful, we need to be able to a) store the projection of the graph in small space and b) solve the problem of interest given only the projection of the graph. While linear sketching is a classic technique for solving statistical problems in the data stream model, it was long thought unlikely to be useful in the context of combinatorial problems on graphs. Not only do linear sketches allow us to process edge deletions (a deletion can just be viewed as a "negative" insertion) but the linearity of the resulting data structures enables a rich set of algorithmic operations to be performed after the sketch has been generated. In fact, it has been shown that any dynamic streaming algorithm can be implemented via linear sketches [108]. Linear sketches are also a useful technique for reducing communication when processing distributed graphs. For a recent survey of graph streaming and sketching see [114].

Graph streams can also be modeled with different assumptions about the order of arrival of elements in the stream. There are several variants: in the *arbitrary order model*, the stream consists of the edges of the graph in arbitrary order. In the *adjacency list model*, all edges that include the same node are grouped together. In the *random order model*, the order in which the edges arrive in stream is chosen uniformly at random from all possible orderings. Both the arbitrary order model and the adjacency list model generalize naturally to hypergraphs where each edge could consists of more than two nodes. The arbitary order model has been more heavily studied than the adjacency list model but there has still been a significant amount of work in the latter model [14, 15, 26, 78, 88, 103, 116–118]. For further details, see a recent survey on work on the graph stream model [114].

In Chapter 4, we introduce the notion of a *temporal* graph stream, which defines a temporal graph via a sequence of edge updates. A temporal graph $T = (V, A)$, $A \subset V \times V \times \mathbb{N}$ in the streaming setting is defined by a sequence of edge insertions $a \in A$ where each edge update contains a timestamp indicating the time at which the edge appeared or disappeared from the temporal graph. In this thesis, we consider insert-only temporal graph streams.

1.1.1 Preliminaries and Notation

Graphs Preliminaries. A hypergraph is specified by a set of vertices $V = \{v_1, \ldots, v_n\}$ and a set of subsets of V called hyperedges. In Chapter 2 and in parts of Chapter 3 we assume all hyperedges have cardinality at most d for some constant d. The special case when all hyperedges have cardinality exactly two corresponds to the standard definition of a graph. All graphs and hypergraphs discussed in this dissertation will be undirected except when specified otherwise. It will be convenient to define the following notation: Let $\delta_G(S)$ be the set of hyperedges that cross the cut $(S, V \setminus S)$ in the hypergraph G where we say a hyperedge e crosses $(S, V \setminus S)$ if $e \cap S \neq \emptyset$ and $e \cap (V \setminus S) \neq \emptyset$. For any hyperedge e, define $\lambda_e(G)$ to be the minimum cardinality of a cut that includes e. A *spanning graph* $H = (V, E)$ of a hypergraph $G = (V, E)$ is a subgraph such that $|\delta_H(S)| \geq \min(1, |\delta_G(S)|)$ for every $S \subset V$.

Linear Sketches and Applications. Many of the streaming algorithms in this thesis use linear sketches.

Definition 1 (Linear Sketches)**.** *A* linear measurement *of a hypergraph on n vertices is defined by a set of coefficients $\{c_e : e \in \mathcal{P}_r(V)\}$ where $\mathcal{P}_r(V)$ is the set of all subsets of V of size at most d. Given a hypergraph $G = (V, E)$, the evaluation of this measurement is defined as $\sum_{e \in E} c_e$. A* sketch *is a collection of (non-adaptive) linear measurements. The cardinality of this collection is referred to as the* size *of the sketch. We will assume that the magnitude of the coefficients c_e is $\mathrm{poly}(n)$. We say a linear measurement is* local *for node v if the measurement only depends on hyper-edges incident to v, i.e., $c_e = 0$ for all hyper-edges that do not include v. We say a sketch is* vertex-based *if every linear measurement is local to some node.*

Linear sketches have long been used in the context of data stream models because it is possible to maintain a sketch of the stream incrementally. Specifically, if the next stream update is an insertion or deletion of an edge, we can update the sketch by simply adding or subtracting the appropriate set of coefficients. Sketches are also useful in distributed settings. In particular, the model considered by Becker et al. [19] was as follows: suppose there are $n + 1$ players $P_1, \ldots, P_n$ and Q. The input for player P_i is the set of (hyper-)edges that include the ith vertex of a graph G. Player Q wants to compute something about this graph such as determining whether G connected. To enable this, each of the players $P_1, \ldots, P_n$ simultaneously sends a message about their input to Q such that the set of these n messages contains sufficient information to complete Q's computation. In the case of randomized protocols, we assume that all players have access to public random bits. The goal is to minimize the maximum length of the n messages that are sent to Q. If a vertex-based sketch exists for the problem under consideration, then for each linear measurement, there is a single player that can evaluate this message and send it to Q.

1.1.2 Connectivity Results in Dynamic (Hyper-)Graph Streams

In Chapter 2, we present sketch-based dynamic graph algorithms for three basic graph problems: computing vertex connectivity, graph reconstruction, and hypergraph sparsification. All our algorithms run in (low) polynomial time, typically linear in the number of edges. However, our primary focus is on space complexity, as is the convention in much of the data streams literature.

Vertex Connectivity. To date, the main success story for graph sketching has been about edge connectivity, i.e., estimating how many edges need to be removed to disconnect the graph, and estimating the size of cuts. We present the first dynamic graph stream algorithms for vertex connectivity, i.e., estimating how many *vertices* need to be removed to disconnect the graph. While it can be shown that edge connectivity is an upper bound for vertex connectivity, the vertex connectivity of a graph can be much smaller. Furthermore, the combinatorial structure relevant to both quantities is very different. For example, edge-connectivity is transitive[1] whereas vertex-connectivity is not. A celebrated result by Karger [95] bounds the number of near minimum cuts whereas no analogous bound is known for vertex removal. Feige et al. [60] discuss issues that arise specific to vertex connectivity in the context of approximation algorithms and embeddings.

[1] If it takes at least k edge deletions to disconnect u and v and it takes at least k edge deletions to disconnect v and w, then it takes at least k edge deletions to disconnect u and w.

In Section 2.1, we present two sketch-based algorithms for vertex connectivity. The first algorithm uses $O(kn \operatorname{polylog} n)$ space and constructs a data structure such that, at the end of the stream, it is possible to test whether the removal of a queried set of at most k vertices would disconnect the graph. We prove that this algorithm is optimal in terms of its space use. The second algorithm estimates the vertex connectivity up to a $(1+\epsilon)$ factor using $O(\epsilon^{-1}kn \operatorname{polylog} n)$ space where k is an upper bound on the vertex connectivity.

No stream algorithms were previously known that supported both edge insertions and deletions. Existing approaches either use $\Omega(n^2)$ space [140] or only handle insertions [56]. With only insertions, Eppstein et al. [56] proved that $O(kn \operatorname{polylog} n)$ space was sufficient. Their algorithm drops an inserted edge $\{u, v\}$ iff there already exists k vertex-disjoint paths between u and v amongst the edges stored thus far. Such an algorithm fails in the presence of edge deletions since some of the vertex disjoint paths that existed when an edge was ignored need not exist if edges are subsequently deleted.

Graph Reconstruction. Our next result relates to reconstructing graphs rather than estimating properties of the graph. Becker et al. [19] show that is possible to reconstruct a μ-degenerate graph (that is, a graph for which all induced subgraphs have a vertex of degree at most μ) given an $O(\mu \operatorname{polylog} n)$ size sketch of each row of the adjacency matrix of the graph. In Section 2.2, we define the μ-cut-degeneracy and show that the strictly larger class of graphs that satisfy this property can also be reconstructed given an $O(\mu \operatorname{polylog} n)$-size sketch of each row. Moreover, even if the graph is not μ-cut-degenerate we show that we can find all edges with a certain connectivity property. This will be an integral part of our algorithm for hypergraph sparsification. For this purpose, we also prove the first dynamic graph stream algorithms for hypergraph connectivity in this section. We also extend the vertex connectivity results to hypergraphs.

Hypergraph Sparsification. Hypergraph sparsification is a natural extension of graph sparsification. Given a hypergraph, the goal is to find a sparse weighted subgraph such that the weight of every cut in the subgraph is within a $(1+\epsilon)$ factor of the weight of the corresponding cut in the original hypergraph. Estimating hypergraph cuts has applications in video object segmentation [81], network security analysis [148], load balancing in parallel computing [29], and modelling communication in parallel sparse-martix vector multiplication [28].

Kogan and Krauthgamer [101] recently presented the first stream algorithm for hypergraph sparsification in the insert-only model. In Section 2.3, we present the first algorithm that supports both edge insertions and deletions. The algorithm uses $O(n \operatorname{polylog} n)$ space assuming that size of the hyperedges is bounded by a constant. This result is part of a growing body of work on processing hypergraphs in the data stream model [54, 101, 133, 138, 144]. There are numerous challenges in extending previous work on graph sparsification [7, 8, 75, 93, 94] to hypergraph sparsification and we discuss these in Section 2.3. In the process of overcoming these challenges, we also identify a simpler approach for graph sparsification in the data stream model.

1.1.3 Coverage Results in Data Streams

In Chapter 3, we present algorithms for the Max-k-Cover and Max-k-UniqueCover problems in the data stream model. The input to both problems are m subsets of a universe of size n and a value $k \in [m]$. In Max-k-Cover, the problem is to find a collection of at most k sets such that the number of elements covered by at least one set is maximized. In Max-k-UniqueCover, the problem is to find a collection of at most k sets such that the number of elements covered by exactly one set is maximized. These problems are closely related to a range of graph problems including matching, partial vertex cover, and capacitated maximum cut.

In the stream model, we assume k is given and the sets are revealed online. Our goal is to design single-pass algorithms that use space that is sublinear in the input size. The following algorithms are for insert-only streams except where specified otherwise. Our main algorithmic results are as follows.

- If sets have size at most d, there exist single-pass algorithms using $O(d^{d+1}k^d)$ space that solve both problems exactly. This is optimal up to logarithmic factors for constant d.

- If each element appears in at most r sets, we present single pass algorithms using $\tilde{O}(k^2r/\epsilon^3)$ space that return a $1+\epsilon$ approximation in the case of Max-k-Cover and $2+\epsilon$ approximation in the case of Max-k-UniqueCover. We also present a single-pass algorithm using slightly more memory, i.e., $\tilde{O}(k^3r/\epsilon^4)$ space, that $1+\epsilon$ approximates Max-k-UniqueCover.

In contrast to the above results, when d and r are arbitrary, any constant pass $1+\epsilon$ approximation algorithm for either problem requires $\Omega(\epsilon^{-2}m)$ space but a single pass $O(mk/\epsilon^2)$ space algorithm exists. In fact any constant-pass algorithm with an approximation better than $e^{1-1/k}$ requires $\Omega(m/k^2)$ space when d and r are unrestricted. En route, we also obtain an algorithm for the parameterized version of the streaming SetCover problem.

Relationship to Graph Streaming.

To explore the relationship between Max-k-Cover and Max-k-UniqueCover and various graph stream problems, it makes sense to introduce to additional parameters beyond m (the number of sets) and n (the size of the universe). Specifically, throughout the chapter we let d denote the maximum cardinality of a set in the input and let r denote the maximum multiplicity of an element in the universe where the *multiplicity* is the number of sets the element appears.[2] Then an input to Max-k-Cover and Max-k-UniqueCover can define a (hyper)graph in one of the following two natural ways:

(1) *First Interpretation:* A sequence of (hyper-)edges on a graph with n nodes of maximum degree r (where the degree of a node v corresponds to how many hyperedges include that node) and m hyperedges where each hyperedge has size at most d. In the

[2]Note that d and r are dual parameters in the sense that if the input is $\{S_1, \ldots, S_m\}$ and we define $T_i = \{j : i \in S_j\}$ then $d = \max_j |S_j|$ and $r = \max_i |T_i|$.

case where every set has size $d = 2$, the hypergraph is an *ordinary graph*, i.e., a graph where every edge just has two endpoints. With this interpretation, the graph is being presented in the arbitrary order model.

(2) *Second Interpretation:* A sequence of adjacency lists (where the adjacency list for a given node includes all the hyperedges) on a graph with m nodes of maximum degree d and n hyperedges of maximum size r. In this interpretation, if every element appears in exactly $r = 2$ sets, then this corresponds to an ordinary graph where each element corresponds to an edge and each element corresponds to an edge. With this interpretation, the graph is being presented in the adjacency list model.

Under the first interpretation, the Max-k-Cover problem and the Max-k-UniqueCover problem when all sets have exactly size 2 naturally generalize the problem of finding a maximum matching in an ordinary graph in the sense that if there exists a matching with at least k edges, the optimum solution to either Max-k-Cover and Max-k-UniqueCover will be a matching. There is a large body of work on graph matchings in the data stream model [5, 27, 47, 48, 57, 63, 74, 76, 89, 90, 102–104, 113, 150] including work specifically on solving the problem exactly if the matching size is bounded [38, 40]. More precisely, Max-k-Cover corresponds to the partial vertex cover problem [111]: what is the maximum number of edges that can be covered by selecting k nodes. For larger sets, the Max-k-Cover and Max-k-UniqueCover are at least as hard as finding partial vertex covers and matching in hypergraphs.

Under the second interpretation, when all elements have multiplicity 2, Max-k-UniqueCover corresponds to finding the capacitated maximum cut, i.e., a set of at most k vertices such that the number of edges with exactly one endpoint in this set is maximized. In the offline setting, Ageev and Sviridenko [4] and Gaur et al. [68] presented a 2 approximation for this problem using linear programming and local search respectively. The (uncapacitated) maximum cut problem was been studied in the data stream model by Kapralov et al. [91,92]; a 2-approximation is trivial in logarithmic space[3] but improving on this requires space that is polynomial in the size of the graph. The capacitated problem is a special case of the problem of maximizing a non-monotone sub-modular function subject to a cardinality constraint. This general problem has been considered in the data stream model [16, 32, 35, 80] but in that line of work it is assumed that there is oracle access to the function being optimized, e.g., given any set of nodes, the oracle will return the number of edges cut. Alaluf et al. [9] presented a $2 + \epsilon$ approximation in this setting, assuming exponential post-processing time. In contrast, our algorithm does not assume an oracle while obtaining a $1 + \epsilon$ approximation (and also works for the more general problem Max-k-UniqueCover).

1.1.4 Temporal Graph Streams

Graphs are extremely general and useful structures which can elegantly represent many aspects of real-world structures such as social networks, disease spreading models, and the Internet. However, one aspect of all of these structures that the traditional view of graphs does not accomodate is temporality. For example, consider disease-tracking on a real-time

[3] It suffices to count the number of edges M since there is always a cut whose size is between $M/2$ and M.

stream of physical contact events between people. Say Alice has the flu. She shakes Bob's hand, and then Bob later shakes Charlie's hand. We could attempt to model this with a graph G with nodes $\{A, B, C\}$ and edges $\{(A, B), (B, C)\}$ where Alice is represented by node A, Bob by node B, Charlie by node C, and edges represent infection-spreading handshakes. We can use this graph to conclude that Charlie is susceptible to infection because there is a path from Alice to him. However, this graph representation would also suggest that if Charlie was the one who started with the flu, Alice is susceptible to infection since there is a path from Charlie to Alice as well. This is incorrect, since Alice interacts with Bob before he can be infected by Charlie. By not taking the *time* at which these edges occurred into account, this graph representation fails to adequately represent the spread of disease. We would like some model that allows us to determine who is at risk of infection from some patient zero, perhaps through an indirect chain of time-ordered contact events. To capture such dynamics, we use *temporal* graphs whose edges are augmented with set of timestamps which indicate times at which the edge exists. In our example, we replace G with temporal graph T with nodes $\{A, B, C\}$ and edges $\{(A, B, 1), (B, C, 2)\}$ where edges are now triples: a pair of endpoints and a timestamp. Note that the path from A to C uses edges with increasing timestamps, suggesting infection can spread along this path, while the path from C to A uses edges with decreasing timestamps, ruling out the possibility of infection spreading in the reverse direction. We say that there is a *time-respecting* path from A to C but not from C to A.

Mertzios et al. [122] give an algorithm that computes short time-respecting paths from a source node s to all other nodes in $O(n \operatorname{poly}(\tau))$ time where τ denotes the number of distinct timestamps in the temporal graph. Menger's theorem, which states that the maximum number of node-disjoint s to t paths is equal to the minimum number of nodes that must be removed in order to separate s from z [121], does not hold for time-respecting paths in temporal graphs [23, 99]. However, Mertzios et. al. [122] recently proved a reformulated temporal analogue of Menger's theorem that holds for temporal graphs.

Researchers have noted that temporal analogues of graph problems tend to have higher complexity. Bhadra & Ferreira [24] demonstrate that computing strongly connected components of directed temporal graphs is NP-Complete. Michail & Spirakis [124] show that a temporal analogue of the maximum matching problem, where one must find a maximum matching whose edges have distinct timestamps, is NP-Complete as well. They also prove that a temporal analogue of the Graphic Traveling Salesman Problem cannot be approximated within multiplicative factor cn for some constant $c > 0$ unless $P = NP$. For the standard and more general TSP, its temporal analogue is APX-Hard even if its edge costs are constrained to $\{1, 2\}$.

The study of temporal graphs is in its infancy [123] and to date no one has considered algorithms on temporal graphs in the streaming domain. It will be useful to study these structures at scale. For instance, in the spirit of our infection example, we may wish to track the spread of disease through a large, densely connected population. What can we accomplish by storing a small summary of the massive stream of connection events?

We begin the study of temporal graph streams by considering variations of reachability problems, which involve determining whether or not there exists a time-respecting path between nodes in the temporal graph. We demonstrate strong lower bounds for many versions of this problem, but also find several versions that admit space-efficient algorithms.

We also present some conjectures about the overall hardness of streaming temporal graph reachability.

1.2 Graph Algorithms for Systems Challenges

A significant portion of the work in this dissertation consists of applications of graph algorithms to practical systems challenges, resulting in open-source software which we show both analytically and empirically to be effective and efficient. In this thesis, we present two such completed projects: MESH, a memory manager that is capable of memory compaction in C and C++ (a feat long thought impossible), and PathCache, an efficiently scalable network measurement platform that outperforms the current state of the art.

1.2.1 Memory Compaction Powered by Graph Algorithms

Memory consumption is a serious concern across the spectrum of modern computing platforms, from mobile to desktop to datacenters. For example, on low-end Android devices, Google reports that more than 99% of Chrome crashes are due to running out of memory when attempting to display a web page [79]. On desktops, the Firefox web browser has been the subject of a five-year effort to reduce its memory footprint [142]. In datacenters, developers implement a range of techniques from custom allocators to other *ad hoc* approaches in an effort to increase memory utilization [135, 139].

A key challenge is that, unlike in garbage-collected environments, automatically reducing a C/C++ application's memory footprint via compaction is not possible. Because the addresses of allocated objects are directly exposed to programmers, C/C++ applications can freely modify or hide addresses. For example, a program may stash addresses in integers, store flags in the low bits of aligned addresses, perform arithmetic on addresses and later reference them, or even store addresses to disk and later reload them. This hostile environment makes it impossible to safely relocate objects: if an object is relocated, all pointers to its original location must be updated. However, there is no way to safely update *every* reference when they are ambiguous, much less when they are absent.

Existing memory allocators for C/C++ employ a variety of best-effort heuristics aimed at reducing average fragmentation [86]. However, these approaches are inherently limited. In a classic result, Robson showed that all such allocators can suffer from catastrophic memory fragmentation [137]. This increase in memory consumption can be as high as the $\log$ of the ratio between the largest and smallest object sizes allocated. For example, for an application that allocates 16-byte and 128KB objects, it is possible for it to consume $13\times$ more memory than required.

Chapter 5 introduces MESH, a plug-in replacement for `malloc` that, for the first time, eliminates fragmentation in unmodified C/C++ applications. MESH combines novel randomized algorithms with widely-supported virtual memory operations to provably reduce fragmentation, breaking the classical Robson bounds with high probability. We focus here on the randomized graphalgorithms which power MESH and proofs of their solution quality and runtime. Because MESH operates on live memory contents of active programs, it operates under extreme time pressure and in essence cannot afford to observe every edge

in the graph. Instead it must find a solution by making a limited number of edge queries which take valuable time to answer. MESH generally matches the runtime performance of state-of-the-art memory allocators while reducing memory consumption; in particular, it reduces the memory of consumption of Firefox by 16% and Redis by 39%.

1.2.2 Efficient Network Measurement via Graph Discovery

Despite its engineered nature, the Internet has evolved into a collection of networks with different–and sometimes conflicting–goals, where understanding its behavior requires empirical study of topology and network paths. This problem is compounded by networks' desire to keep their routing policies and behaviors opaque to outsiders for commercial or security-related reasons. Researchers have worked for over a decade designing tools and techniques for inferring AS level connectivity and paths [109, 110]. However, operators seeking to leverage information about network paths or researchers requiring Internet paths to evaluate new Internet-scale systems are often confronted with limited vantage points for direct measurement and myriad data sets, each offering a different lens on AS level connectivity and paths.

Predicting network paths is crucial for a variety of problems impacting researchers, network operators and content/cloud providers. Researchers often need knowledge of Internet routing to evaluate Internet-scale systems (e.g., refraction routing [146], Tor [51], Secure-BGP [71]). For network operators, network paths can aid in diagnosing the root cause of performance problems like high latency or packet loss. Content providers debugging poor client-side performance require the knowledge of the set of networks participating in delivering client traffic to root-cause bottleneck links. While large cloud providers, like Amazon, Google and Microsoft are known to develop in-house telemetry for global network diagnostics, small companies, ISPs and academics often lack such visibility and data.

Understanding and predicting Internet routes is confounded by several factors. Internet paths are dependant on several deterministic but not public phenomena: route advertisements made via BGP and best path selection algorithms based on private business relationships. Additionally, factors like load balancing via ECMP, intermittent congestion on network links, control plane mis-configurations and BGP hijacks also impact network paths.

Standard diagnostic tools like traceroute provide limited visibility into network paths since the user can only control the destination of a traceroute query, the source being her own network. Tools like reverse traceroute [97] rely on the support of IP options to shed light on the reverse path towards one's network. In addition to requiring the support for IP options from Internet routers, these techniques require active probing from the client (or a set of vantage points distributed on the Internet). Active probing is not only expensive in terms of amount of traffic generated (traceroutes, pings etc.) but also provides limited visibility into the network state.

In Chapter 6, we design and develop PathCache, which predicts network paths between arbitrary sources and destinations on the Internet by developing probabilistic models of routing from observed network paths. For this purpose, PathCache, leverages existing data and control plane measurements (such as stale traceroutes and BGP routing data), optimizing use of existing data plane measurement platforms, and applying routing models when empirical data is absent. Specifically, the challenge is to select a bounded number of

path measurement queries to make towards some destination which maximize the amount of network topology (modeled as a directed graph with the destination as the ”root”) discovered. Using provably efficient algorithms, PathCache consumes millions of traceroutes from public measurement platforms every hour and updates the probabilistic routing model using newly acquired information. We offer PathCache as a service at `https://www.davidtench.com/deeplinks/pathcache`. In its present form, the PathCache REST API allows users to query network paths between sources and destinations (IP address, BGP routed prefix or autonomous system). In addition to providing the predicted paths, PathCache provides *confidence* values associated with each network path based on historical information.

PathCache complements the approach of existing path-prediction systems [98, 109] by developing efficient algorithms for measuring the routing behavior towards all BGP prefixes on the Internet. When measuring paths towards each BGP prefix, PathCache maximizes discovery of the network topology with a constrained measurement budget, both globally and per vantage point. PathCache's strategy for exploring network paths discovers 4X more AS-hops than other well known strategies used in practice today.

Problems in the Graph Streaming Model and Extensions

CHAPTER 2

VERTEX AND HYPEREDGE CONNECTIVITY IN DYNAMIC GRAPH STREAMS

A growing body of work addresses the challenge of processing dynamic graph streams: a graph is defined by a sequence of edge insertions and deletions and the goal is to construct synopses and compute properties of the graph while using only limited memory. Linear sketches have proved to be a powerful technique in this model and can also be used to minimize communication in distributed graph processing.

We present the first linear sketches for estimating vertex connectivity and constructing hypergraph sparsifiers. Vertex connectivity exhibits markedly different combinatorial structure than edge connectivity and appears to be harder to estimate in the dynamic graph stream model. Our hypergraph result generalizes the work of Ahn et al. [6] on graph sparsification and has the added benefit of significantly simplifying the previous results. One of the main ideas is related to the problem of reconstructing subgraphs that satisfy a specific sparsity property. We introduce a more general notion of graph degeneracy and extend the graph reconstruction result of Becker et al. [19].

2.1 Vertex Connectivity

A natural approach to determining vertex connectivity could be to try to mimic the algorithm of Cheriyan et al. [36]. They showed that the union of k disjoint "scan first search trees" (a generalization of breadth-first search trees) can be used to determine if a graph is k vertex connected. A similar approach worked in data stream model for the case of edge-connectivity (which we discuss in further detail in the next section) but in that case the trees to be constructed could be arbitrary. Unfortunately, we can show that any algorithm for constructing a scan-first search tree in the data stream model requires $\Omega(n^2)$ space even when there are no edge deletions.

A scan first search tree (SFST) of a graph [36] is defined as follows: The tree is initially empty, all vertices except the root (chosen arbitrarily) are *unmarked*, and all vertices are *unscanned*. At each step we *scan* an marked but unscanned vertex. For each vertex x that is being scanned, all edges from x to unmarked neighbors of x are added to the tree and the unmarked neighbors are marked. This continues until no marked but unscanned vertices remain.

Theorem 2. *Any data stream algorithm that constructs a SFST with probability at least* $3/4$ *requires* $\Omega(n^2)$ *space.*

Proof. The proof is by a reduction from the communication problem of indexing [1]. Suppose Alice has a binary string $x \in \{0,1\}^{n^2}$ indexed by $[n] \times [n]$ and Bob wants to compute $x_{i,j}$ for some index $(i,j) \in [n] \times [n]$ that is unknown to Alice. This requires $\Omega(n^2)$ bits to be communicated from Alice to Bob if Bob is to learn $x_{i,j}$ with probability at least $3/4$. Suppose we have a data stream algorithm for constructing an SFST. Alice creates a graph on nodes $T \cup U \cup V \cup W$ where $T = \{t_1, \ldots, t_n\}, U = \{u_1, \ldots, u_n\}, V = \{v_1, \ldots, v_n\}$, and $W = \{w_1, \ldots, w_n\}$. She adds edges $\{t_k, u_\ell\}$ and $\{v_\ell, t_k\}$ for each ℓ, k such that $x_{\ell,k} = 1$. Alice runs the scan-first search algorithm and sends the contents of her memory to Bob. Bob adds the edge $\{u_i, v_i\}$. Note that any SFST includes all neighbors of u_i or v_i. In particular, $x_{i,j} = 1$ iff at least one of $\{t_j, u_i\}$ or $\{v_i, w_j\}$ is present in the SFST constructed. Hence, the algorithm must have used $\Omega(n^2)$ space. □

To avoid this issue, we take a different approach based on finding arbitrary spanning trees for the induced graph on a random subset of vertices.[1] We will use the following result for finding these spanning trees.

Theorem 3 (Ahn et al. [6])**.** *For a graph on n vertices, there exists a vertex-based sketch of size $O(n \operatorname{polylog} n)$ from which we can construct a spanning forest with high probability.*

Note that in this section we restrict our attention to graphs rather than hypergraphs. However, in the next section we will explain how the vertex connectivity results extend to hypergraphs.

2.1.1 Warm-Up: Vertex Connectivity Queries

For $i = 1, 2, \ldots, R := 16 \cdot k^2 \ln n$, let G_i be a graph formed by deleting each vertex in G with probability $1 - 1/k$. Let T_i be an arbitrary spanning forest of G_i and define $H = T_1 \cup T_2 \cup \ldots \cup T_R$.

Lemma 4. *Let S be an arbitrary collection of at most k vertices. With high probability, $H \setminus S$ is connected iff $G \setminus S$ is connected.*

Proof. First we note that H has the same set of vertices as G with high probability. This follows because the probability a given vertex is not in H is $(1 - 1/k)^R \leq \exp((\,) - 16 \cdot k \cdot \ln n) = n^{-16k}$ and hence by an application of the union bound, all vertices in G are also in H with probability at least $1 - n^{-(16k-1)}$. Then since H is a subgraph of G, then $G \setminus S$ disconnected implies $H \setminus S$ disconnected. It remains to prove that $G \setminus S$ connected implies $H \setminus S$ connected.

Assume $G \setminus S$ is connected. Consider an arbitrary pair of vertices $s, t \notin S$ and let $s = v_0 \to v_1 \to v_2 \to \ldots \to v_\ell = t$ be a path between s and t in $G \setminus S$. Then note that there is a path between v_i and v_{i+1} in $H \setminus S$ if there exists G_i such that $G_i \cap S = \emptyset$ and

[1] We note that the idea of subsampling vertices was recently explored by Censor-Hillel et al. [30, 31]. They showed that if each vertex of a k-vertex-connected graph is subsampled with probability $p = \Omega(\sqrt{\log n / k})$ then the resulting graph has vertex connectivity $\Omega(kp^2)$. We do not make use of this result in our work as it does not lead to an approximation factor better than $\sqrt{k}$.

$v_i, v_{i+1} \in H \setminus S$. This follows because if $\{v_i, v_{i+1}\} \in G_i$ and $G_i \cap S = \emptyset$ then $T_i \setminus S$ either contains $\{v_i, v_j\}$ or a path between between v_i and v_j. Hence,

$$\mathbb{P}\left[v_i \text{ and } v_{i+1} \text{ are connected in } T_i \setminus S\right] \geq 1/k^2(1-1/k)^k$$

and therefore

$$\mathbb{P}\left[v_i \text{ and } v_{i+1} \text{ are disconnected in } T_i \setminus S \text{ for all } i \in [R]\right] \leq (1-1/k^2(1-1/k)^k)^R \leq 1/n^4 .$$

Taking the union bound over all $\ell < n$ pairs $\{v_i, v_{i+1}\}$, we conclude that s and t are connected in $H \setminus S$ with probability at least $1 - 1/n^3$. By applying the union bound again, with probability at least $1 - 1/n^2$, s is connected in $H \setminus S$ to all other vertices. □

Our algorithm constructs a spanning forest for each of $G_1, \ldots, G_R$ using the algorithm referenced in Theorem 3. Note that since each G_i has $O(n/k)$ vertices with high probability, we can construct these R trees in $R \cdot O(n/k \operatorname{polylog} n) = O(nk \operatorname{polylog} n)$ space. This gives us the following theorem.

Theorem 5. *There is a sketch-based dynamic graph algorithm that uses $O(kn \operatorname{polylog} n)$ space to test whether a set of vertices S of size at most k disconnects the graph. The query set S is specified at the end of the stream.*

We next prove that the above query algorithm is space-optimal.

Theorem 6. *Any dynamic graph algorithm that allows us to test, with probability at least $3/4$, whether a queried set of at most k vertices disconnects the graph requires $\Omega(kn)$ space.*

Proof. The proof is by a reduction from the communication problem of indexing [1]. Suppose Alice has a binary string $x \in \{0,1\}^{(k+1)\cdot n}$ indexed by $[k+1] \cdot [n]$ and Bob wants to compute $x_{i,j}$ for some index $(i,j) \in [k+1] \cdot [n]$ that is unknown to Alice. This requires $\Omega(nk)$ bits to be communicated from Alice to Bob if Bob is to be successful with probability at least $3/4$. Consider the protocol where the players create a bipartite graph on vertices $\mathcal{L} \cup \mathcal{R}$ where $\mathcal{L} = \{l_1, \ldots, l_{k+1}\}$ and $\mathcal{R} = \{r_1, \ldots, r_n\}$. Alice adds edges $\{l_i, r_j\}$ for all pairs (i,j) such that $x_{i,j} = 1$. Alice runs the algorithm and sends the state to Bob. Bob adds edges $\{r_\ell, r_{\ell'}\}$ for all $\ell, \ell' \neq j$ and deletes all vertices in L except l_i. Now r_j is connected to the rest of the graph iff the $x_{i,j} = 1$. □

2.1.2 Vertex Connectivity

For $i = 1, 2, \ldots, R := 160 \cdot k^2 \epsilon^{-1} \ln n$, let G_i be a graph formed by deleting each vertex in G with probability $1 - 1/k$. As before, let T_i be an arbitrary spanning forest of G_i and define $H = T_1 \cup T_2 \cup \ldots \cup T_R$.

Theorem 7. *Let S be a subset of V of size k. Consider any pair of vertices $u, v \in V \setminus S$ such that there are at least $(1+\epsilon)k$ vertex-disjoint paths between u and v in G. Then,*

$$\mathbb{P}\left[u \text{ and } v \text{ are connected in } G_S\right] \geq 1 - 4/n^{10k}$$

where $G_S = \cup_{i \in U(S)} G_i$ and $U(S) = \{i : G_i \cap S = \emptyset\}$ is the set of sampled graphs with no vertices in S.

Proof. We first argue that $|U(S)|$ is large with high probability. Then $\mathbb{E}[|U(S)|] = (1 - 1/k)^k R \geq R/4$. By an application of the Chernoff bound:

$$\mathbb{P}[|U(S)| \leq 1/2 \cdot R/4] \leq e^{-1/4 \cdot R/4 \cdot 1/3} < 1/n^{10k} .$$

In the rest of the proof we condition on event $|U(S)| \geq r := R/8$.

Note that there are $t \geq \epsilon k$ vertex-disjoint paths between u and v in $G \setminus S$. Call these paths $P_1, \ldots, P_t$. For each P_i, let a_i be the edge incident to u, let c_i be the edge incident to v, and let B_i be the remaining edges in P_i. Note that a_i and c_i need not be distinct and B_i could be empty.

Claim 1. *The followings three probabilities are each larger than* $1 - 1/n^{10k}$*:*

$$\mathbb{P}[a_i \in G_S \textit{ for at least } 3t/4 \textit{ values of } i]$$

$$\mathbb{P}[B_i \subseteq G_S \textit{ for at least } 3t/4 \textit{ values of } i]$$

$$\mathbb{P}[c_i \in G_S \textit{ for at least } 3t/4 \textit{ values of } i] .$$

Proof. Each edge in B_i is not present in G_S with probability $(1 - 1/k^2)^r$. Hence, by the union bound, $\mathbb{P}[B_i \not\subseteq G_S] \leq |B_i|(1 - 1/k^2)^r$. Also by the union bound,

$$\begin{aligned} & \mathbb{P}[B_i \not\subseteq G_S \text{ for more than } t/4 \text{ values of } i] \\ < \; & \binom{t}{t/4} (n(1 - 1/k^2)^r)^{t/4} \\ < \; & \exp\left(t \ln 2 + (\ln n - r/k^2)t/4\right) < 1/n^{10k} . \end{aligned}$$

The proofs for a_i and c_i are entirely symmetric so we just consider a_i. Consider the set $U'(S) = U(S) \cap \{j : u \in G_j\}$. Note that for $j \in U'(S)$ we have $\mathbb{P}[a_i \in G_j] = 1/k$ and by the union bound,

$$\begin{aligned} & \mathbb{P}\left[a_i \notin \cup_{j \in U'(S)} G_j \text{ for at least } t/4 \text{ values of } i\right] \\ \leq \; & \binom{t}{t/4} (1 - 1/k)^{|U'(S)|t/4} \\ \leq \; & 2^t \exp\left(\frac{-|U'(S)|t}{(4k)}\right) . \end{aligned}$$

Let E be the event that $|U'(S)| \leq |U(S)|/(2k)$. Then, by an application of the Chernoff bound:

$$\begin{aligned} & \mathbb{P}[a_i \notin G_S \text{ for at least } t/4 \text{ values of } i] \\ \leq \; & \mathbb{P}[E] \\ & + \mathbb{P}\left[a_i \notin \cup_{j \in U'(S)} G_j \text{ for at least } t/4 \text{ values of } i \mid \neg E\right] \\ \leq \; & \exp\left(-1/4 \cdot |U(S)|/k \cdot 1/3\right) \\ & \quad + \mathbb{P}\left[a_i \notin \cup_{j \in U'(S)} G_j \text{ for at least } t/4 \text{ values of } i \mid \neg E\right] \\ \leq \; & \exp\left(-1/4 \cdot r/k \cdot 1/3\right) + 2^t \exp\left(-r/(2k) \cdot t/(4k)\right) \\ < \; & 1/n^{10k} . \end{aligned}$$

□

It follows from the claim that there exists i such that $P_i \in G_S$ (and therefore u and v are connected in G_S) with probability at least $1 - 3/n^{10k}$. The conditioning on $|U(S)| \geq r$ decreases this by another $1/n^{10k}$. □

Corollary 8. *If G is $(1+\epsilon)k$-vertex-connected then H is k-vertex-connected with high probability. If H is k-vertex connected then G is k-vertex connected.*

Proof. The first part of the corollary follows from Theorem 7 by applying the union bound over all $O(n^k)$ subsets of size at most k and $O(n^2)$ choices of u and v. Note that u and v connected in G_S implies u and v are connected in H since H includes a spanning forest of G_S. The second part is implied by the fact H is a subgraph of G. □

As in the previous section, our algorithm is simply to construct H be using the algorithm referenced in Theorem 3 to construct $T_1, \ldots, T_R$. We can then run any vertex connectivity algorithm on H in post-processing. Since each G_i has $O(n/k)$ vertices with high probability, we can construct these R trees in $R \cdot O(n/k \cdot \operatorname{polylog} n) = O(nk\epsilon^{-1} \operatorname{polylog} n)$ space. This gives us the following theorem.

Theorem 9. *There is a sketch-based dynamic graph algorithm that uses $O(kn\epsilon^{-1} \operatorname{polylog} n)$ space to distinguish $(1+\epsilon)k$-vertex connected graphs from k-connected graphs.*

2.2 Reconstructing Hypergraphs

We next present sketches for reconstructing cut-degenerate hypergraphs. Recall that a hypergraph is μ-degenerate if all induced subgraphs have a vertex of degree at most μ. Cut-degeneracy is defined as follows.

Definition 10. *A hypergraph is* μ-cut-degenerate *if every induced subgraph has a cut of size at most μ.*

The following lemma establishes that this is a strictly weaker property than μ-degeneracy.

Lemma 11. *Any hypergraph that is μ-degenerate is also μ-cut-degenerate. There exists graphs that are μ-cut-degenerate but not μ-degenerate.*

Proof. Since the degree of a vertex v is exactly the size of the cut $(\{v\}, V \setminus \{v\})$ it is immediate that μ-degeneracy implies μ-cut-degeneracy. For an example that μ-cut-degenerate does not imply it is μ-degenerate consider the graph G on eight vertices $\{v_1, v_2, v_3, v_4, u_1, u_2, u_3, u_4\}$ with edges $\{v_i, v_j\}, \{u_i, u_j\}$ for all i, j except $i = 1, j = 4$ and edges $\{v_1, u_1\}$ and $\{v_4, u_4\}$. Then G has minimum degree 3 and is therefore not 2-degenerate while it is 2-cut-degenerate. □

Becker et al. [19] showed how to reconstruct a μ-degenerate graph in the simultaneous communication model if each player sends an $O(\mu \operatorname{polylog} n)$ bit message. We will show that it is also possible to reconstruct any μ-cut-degenerate with the same message complexity. Even if the graph is not cut-degenerate, we show that is possible to reconstruct all edges with a certain connectivity property. We will subsequently use this fact in Section 2.3.

2.2.1 Skeletons for Hypergraphs

We first review the existing results on constructing k-skeletons [6] that we will need for our new results. In doing so, we generalize the previous work to the case of hypergraphs. In particular, this leads to the first dynamic graph algorithm for determining hypergraph connectivity.

Definition 12 (k-skeleton)**.** *Given a hypergraph $H = (V, E)$, a subgraph $H' = (V, E')$ is a* k-skeleton *of H if for any $S \subset V$, $|\delta_{H'}(S)| \geq \min(|\delta_H(S)|, k)$.*

In particular, any spanning graph is a 1-skeleton and it can be shown that $F_1 \cup F_2 \cup \ldots \cup F_k$ is a k-skeleton [6] of G if F_i is a spanning graph of $G \setminus (\cup_{j=1}^{i-1} F_j)$. The next lemma establishes that given an arbitrary k-skeleton of a graph we can exactly determine the set of edges with $\lambda_e(G) \leq k - 1$.

Lemma 13. *Let H be a k-skeleton of G then $\lambda_e(H) \leq k - 1$ iff $\lambda_e(G) \leq k - 1$.*

Proof. Since H is a subgraph $\lambda_e(H) \leq \lambda_e(G)$ and hence $\lambda_e(G) \leq k - 1$ implies $\lambda_e(H) \leq k - 1$. Using the fact that H is a k-skeleton $\lambda_e(H) \geq \min(k, \lambda_e(G))$ and hence, if $\lambda_e(H) \leq k - 1$ it must be that $\lambda_e(G) \leq k - 1$. □

Constructing Spanning Graphs. For each vertex $v_i \in V$, define the vector $\mathbf{a}^i \in \{-1, 0, 1, 2, \ldots, d-1\}^\alpha$ where $\alpha = \sum_{i=2}^{d} \binom{n}{i}$ is the number of possible hyperedges of size at most d:

$$\mathbf{a}^i_e = \begin{cases} |e| - 1 & \text{if } i = \min e \text{ and } e \in E \\ -1 & \text{if } i \in e \setminus \min e \text{ and } e \in E \\ 0 & \text{otherwise} \end{cases}$$

where e ranges over all subsets of V of size between 2 and d and $\min e$ denotes the smallest ID of a node in e. Observe that these vectors have the property that for any subset of vertices $\{v_i\}_{i \in S}$, the non-zero entries of $\sum_{i \in S} \mathbf{a}^i$ correspond exactly to $\delta(S)$. This follows because the only subsets of

$$\{|e| - 1, \underbrace{-1, -1, \ldots, -1}_{|e|-1}\}$$

that sum to zero are the empty set and the entire set. Hence, the e-th coordinate of $\sum_{i \in S} \mathbf{a}^i$ is zero iff either $e \notin E$ or $e \subset S$ or $e \subset V \setminus S$.

The rest of algorithm proceeds exactly as in the case of (non-hyper) graphs [6] and a reader that is very familiar with the previous work should feel free to skip the remainder of Section 2.2.1. We construct the sketches $M\mathbf{a}^1, \ldots, M\mathbf{a}^n$ where M is chosen according to a distribution over matrices $\mathbb{R}^{k \cdot \alpha}$ where $k = \text{polylog}(\alpha)$. The distribution has the property that for any $\mathbf{a} \in \mathbb{R}^d$, it is possible to determine the index of a non-zero entry of $\mathbf{a}$ given $M\mathbf{a}$ with probability $1 - 1/\text{poly}(n)$. Such as distribution is known to exist by a result of Jowhari et al. [87]. Given $M\mathbf{a}^1, \ldots, M\mathbf{a}^n$ we can find an edge across an arbitrary cut $(S, V \setminus S)$. To do this, we compute $\sum_{i \in S} M\mathbf{a}^i = M(\sum_{i \in S} \mathbf{a}^i)$. We can then determine the index of a non-zero entry of $\sum_{i \in S} \mathbf{a}^i$ which corresponds to an element of $\delta(S)$ as required. It may appear that to test connectivity we need to test all $2^{n-1} - 1$ possible cuts. Since the failure probability for each cut is only inverse polynomial in n this would be problematic. However, it is possible to be more efficient and only test $O(n)$ cuts. See Ahn et al. [6] for details.

Theorem 14 (Spanning Graph Sketches)**.** *There exists a vertex-based sketch $\mathcal{A}$ of size $O(n \operatorname{polylog} n)$ such that we can find a spanning graph of a hypergraph G from $\mathcal{A}(G)$ with high probability.*

Note the above theorem can be substituted for Theorem 3 and the resulting algorithms for vertex connectivity go through for hypergraphs unchanged.

Constructing k-skeletons. As mentioned above, it suffices to find $F_1, \ldots, F_k$ such that F_i is a spanning graph of $G \setminus (\cup_{j=1}^{i-1} F_j)$. Do to this we use k independent spanning graph sketches $\mathcal{A}^1(G), \mathcal{A}^2(G), \ldots, \mathcal{A}^k(G)$ as described in the previous section. We may construct F_1 from $\mathcal{A}^1(G)$ because this is the functionality of a spanning graph sketch. Assuming we have already constructed $F_1, \ldots, F_{i-1}$ we can construct F_i from:

$$\mathcal{A}^i(G - F_1 - F_2 \ldots - F_{i-1}) = \mathcal{A}^i(G) - \sum_{j=1}^{i-1} \mathcal{A}^i(F_j) \ .$$

Theorem 15 (k-Skeleton Sketches)**.** *There exists a vertex-based sketch $\mathcal{B}$ of size $O(kn \operatorname{polylog} n)$ such that we can find of a k-skeleton a hypergraph G from $\mathcal{B}(G)$ with high probability.*

2.2.2 Beyond k-Skeletons

One might be tempted as ask whether it was necessary to use k independent spanning graph sketches $\mathcal{A}^1, \ldots, \mathcal{A}^k$ rather that reuse a single sketch $\mathcal{A}$. If each application of the sketch $\mathcal{A}$ fails to return a spanning graph with probability δ, one might hope to use the union bound to argue that the probability that $\mathcal{A}$ fails on any of the inputs $G, G - F_1, G - F_1 - F_2, \ldots, G - F_1 - \ldots - F_{k-1}$ is at most $k\delta$. ***But this would not be a valid application of the union bound!*** The union bound states that for any *fixed* set of t events $B_1, \ldots, B_t$, we have $\mathbb{P}[B_1 \cup \ldots \cup B_t] \leq \sum_i \mathbb{P}[B_i]$. The issue is that the events in the above example are not fixed, i.e., they can not be specified a priori, since spanning graph F_i is determined by the randomness in the sketch.[2] We belabor this point because, while the union bound was not applicable in the above case, we will need it to prove our next result in a situation that is only subtly different and yet the union bound *is* valid.

2.2.2.1 Finding the light edges

Given a graph $G = (V, E)$ and a postive integer k, recursively define

$$E_i = \{e \in E : \lambda_e(G \setminus \bigcup_{j=1}^{i-1} E_i) \leq k\}$$

[2] Another way to see that using the same sketch cannot work is that if it were possible to repeatedly remove each spanning graph from the sketch of the original graph, we would be able to reconstruct the entire graph using only a sketch of size $O(n \operatorname{polylog} n)$. Clearly this is not possible because it requires at $\Omega(n^2)$ bits to specify an arbitrary graph on n vertices.

and denote the union of these sets as:

$$\text{light}_k(G) = \bigcup_{i \geq 1} E_i \ .$$

Note that if G is μ cut-degenerate then $\text{light}_\mu(G) = E$. Furthermore, there is at most n values of i such that E_i is non-empty since removing each non-empty set E_i from the graph increases the number of connected components.

Suppose $\mathcal{B}(G)$ is a sketch that returns an arbitrary $(k+1)$-skeleton of G with failure probability $\delta = 1/\operatorname{poly}(n)$. Then, since $E_1, E_2, \ldots, E_n$ are sets defined solely by the input graph (and not any randomness in a sketch) we can specify the fixed events

$$\begin{aligned} B_i = \text{“We fail to return a } &(k+1)\text{-skeleton sketch of} \\ &G - E_1 - \ldots - E_i \text{ given } \mathcal{B}(G - E_1 - \ldots - E_i)\text{”} \end{aligned}$$

and therefore use the union bound to establish that the probability that we find a $(k+1)$-skeleton of each of the relevant graphs with failure probability at most $n\delta = 1/\operatorname{poly}(n)$.

We can therefore find the sets $E_1, E_2, \ldots, E_n$ as follows. Let S_i be an arbitrary $(k+1)$ skeleton of $G - E_1 - \ldots E_{i-1}$. Assuming we have already determined $E_1, \ldots, E_{i-1}$, we can find S_i using:

$$\mathcal{B}(G - E_1 - E_2 \ldots - E_{i-1}) = \mathcal{B}(G) - \sum_{j=1}^{i-1} \mathcal{B}(E_j) \ .$$

Then, by appealing to Lemma 13, we know that we can then uniquely determine E_i given S_i.

Theorem 16. *There exists a vertex-based sketch of size $\tilde{O}(kn)$ from which $\text{light}_k(G)$ can be reconstructed for any hypergraph G. In the case of a k-cut-degenerate graph, this is the entire graph.*

2.2.2.2 What are the light edges?

In this section, we restrict our attention to graphs rather than hypergraphs and show that the set of edges in $\text{light}_k(G)$ can be defined in terms of the notion of *strong connectivity* introduced by Benczúr and Karger [20].

Lemma 17. $\text{light}_k(G) = \{e : k_e \leq k\}$ *where $k_{\{u,v\}}$ is the maximum k such that there is a set $S \subset V$ including u and v such that the induced graph on S is k-edge-connected.*

Proof. Define t_e to be the minimum value of k such that $e \in \text{light}_k(G)$. We prove that $t_e = k_e$ and the result follows. To show $k_e \geq t_e$ suppose $t_e = t$ and then note that e survives when we recursively remove edges with edge connectivity $t-1$. But the remaining components in this graph are at least $(t-1)+1 = t$ connected so $k_e \geq t$. To show that $k_e \leq t_e$, suppose $k_e = k$. Then there exists a vertex induced subgraph H containing e that is k-connected. But when we recursively remove edges with edge connectivity at most $k-1$ then no edge in H can be removed. Hence, $t_e > (k-1)$ and so $t_e \geq k$. □

2.3 Hypergraph Sparsification

In this final section, we present a vertex-based sketch for constructing a sparsifier of a hypergraph. This yields the first dynamic graph stream algorithm for constructing a sparsifier of a hypergraph. As an added bonus, our approach gives an algorithm and analysis that is significantly simpler than previous work on the specific case of graph sparsification [7, 75].

Definition 18 (Hypergraph Sparsifier)**.** *A weighted subgraph $H = (V, E', w)$ of a hypergraph $G = (V, E)$ is a* sparsfier *if for all $S \subset V$, $\sum_{e \in \delta_H(S)} w(e) = (1 \pm \epsilon)|\delta_G(S)|$.*

Previous approaches to sparsification in the dynamic stream model relied on work by Fung et al. [67]. To construct a *graph* sparsifier, they showed that it was sufficient to independently sample every edge in the graph with probability $O(\epsilon^{-2}\lambda_e^{-1} \log n)$. Using their work required coopting their machinery and modifying it appropriately (e.g., replacing Chernoff arguments with careful Martingale arguments). Another downside to the previous approach is that the Fung et al. result does not seem to extend to the case of hypergraphs.[3]

Using our new-found ability (see the previous section) to find the entire set of edges that are not k-strong, we present an algorithm that a) has a simpler, and almost self-contained, analysis and b) extends to hypergraphs. Our approach is closer in spirit to Benczúr and Karger's original work on sparsification [20] which in turn is based on the following result by Karger [96]: if we sample each edge with probability $p \geq p^* = c\epsilon^{-2}\lambda^{-1} \log n$ where λ is the cardinality of the minimum cut and $c \geq 0$ is some constant, and weight the sampled edges by $1/p$ then the resulting graph is a sparsifier with high probability.

The idea behind our algorithm is as follows. For a hypergraph G, if we remove the hyperedges $\text{light}_k(G)$ where $k = 2c\epsilon^{-2} \log n$, then every connected component in the remaining hypergraph has minimum cut of size greater than $2c\epsilon^{-2} \log n$. Hence, for each of these components $p^* \leq 1/2$. Therefore, the graph formed by sampling the hyperedges in $G \setminus \text{light}_k(G)$ with probability 1/2 (and doubling the weight of sampled hyperedges) and adding the set of hyperedges in $\text{light}_k(G)$ with unit weights is a sparsifier of G. We then repeat this process until there are no hyperedges left to sample.

Algorithm.

(1) Generate a series of graphs $G_0, G_1, G_2 \ldots$ where G_i is formed by deleting each hyperedge in G_{i-1} independently with probability 1/2 and $G_0 = G$.

(2) For $i = 0, 1, 2, \ldots, \ell = 3 \log n$:

  (a) Let $F_i = \text{light}_k(H_i)$ where $k = O(\epsilon^{-2}(\log n + r))$ where $H_i = G_i \setminus (F_0 \cup F_1 \cup F_2 \cup \ldots \cup F_{i-1})$

(3) Return $\bigcup_{i=0}^{\ell} 2^i \cdot F_i$ where $2^i \cdot F_i$ is the set of hyperedges in F_i where each is given weight 2^i.

Analysis. The following lemma uses an argument due to Karger [95] combined with a hypergraph cut counting result by Kogan and Krauthgamer [101].

[3]For the reader familiar with Fung et al. [67], the issue is finding a suitable definition of cut-projection for hypergraphs and then proving a bound on the number of distinct cut-projections.

Lemma 19. *$2H_{i+1} \cup F_i$ is a $(1+\epsilon)$-sparsifier for H_i.*

Proof. It suffices to prove that $2H_{i+1}$ is a $(1+\epsilon)$-sparsifier for $H_i \setminus F_i$. Furthermore, it suffices to consider each connected component of $H_i \setminus F_i$ separately.

Let C be an arbitrary connected component of $H_i \setminus F_i$ and note that C has a minimum cut of size at least k. Let C' be the graph formed by deleting each hyperedge in C with probability $1/2$. Consider a cut of size t in C and let X be the number of hyperedges in this cut that are in C'. Then $\mathbb{E}[X] = t/2$ and by an application of the Chernoff bound, $\mathbb{P}[|X - t/2| \geq \epsilon t/2] \leq 2\exp(-\epsilon^2 t/6)$.

The number of cuts of size at most t is $\exp(O(dt/k + t/k \cdot \log n))$ by appealing to a result by Kogan and Krauthgamer [101]. By an application of the union bound, the probability that there exists a cut of size t such that the number of hyperedges in the corresponding cut in C' is not $(1 \pm \epsilon)t/2$ is at most

$$2\exp\left(-\epsilon^2 t/6\right) \cdot \exp\left(O(dt/k + t/k \cdot \log n)\right) \ .$$

This probability is less than $1/n^{10}$ if $k \geq c\epsilon^{-2}(\log n + d)$ for some sufficiently large constant c. Hence, taking the union bound over all $t \geq k$ ensures that with probability at least $1/n^8$, for every cut in C, the fraction of edges in the corresponding cut in C' is $(1 \pm \epsilon)/2$. □

Theorem 20. *$\bigcup_{i=0}^{\ell} 2^i \cdot F_i$ is a $(1+\epsilon)^{\ell}$-sparsifier of G where $\ell = 3\log n$.*

Proof. The theorem follows by repeatedly applying Lemma 19. Specifically,

(1) $F_{\ell-1}$ is a $(1+\epsilon)$ sparsifier for $H_{\ell-1}$ since H_ℓ is the empty graph with high probability.

(2) $2H_{\ell-1} \cup F_{\ell-2}$ is a $(1+\epsilon)$-sparsifier for $H_{\ell-2}$ and so $2F_{\ell-1} \cup F_{\ell-2}$ is a $(1+\epsilon)^2$-sparsifier for $H_{\ell-2}$

(3) $2H_{\ell-2} \cup F_{\ell-3}$ is a $(1+\epsilon)$-sparsifier for $H_{\ell-3}$ and so $4F_{\ell-1} \cup 2F_{\ell-2} \cup F_{\ell-3}$ is a $(1+\epsilon)^3$-sparsifier for $H_{\ell-3}$

We continue in this way until we deduce $\bigcup_{i=0}^{\ell} 2^i \cdot F_i$ is a $(1+\epsilon)^{\ell}$-sparsifier for $H_0 = G_0$. □

By re-parameterizing $\epsilon \leftarrow \epsilon/(2\ell)$ and using the sketches from Section 2.2, we establish the next theorem.

Theorem 21. *There exists a vertex-based sketch of size $\tilde{O}(\epsilon^{-2}n)$ from which we can construct a $(1+\epsilon)$ hypergraph sparsifier.*

CHAPTER 3

MAXIMUM COVERAGE IN THE DATA STREAM MODEL: PARAMETERIZED AND GENERALIZED

3.1 Introduction

We consider the Max-k-Cover and Max-k-UniqueCover problems in the data stream model. The input to both problems are m subsets of a universe of size n and a value $k \in [m]$. In Max-k-Cover, the problem is to find a collection of at most k sets such that the number of elements covered by at least one set is maximized. In Max-k-UniqueCover, the problem is to find a collection of at most k sets such that the number of elements covered by exactly one set is maximized. In the stream model, we assume k is provided but that the sets are revealed online and our goal is to design single-pass algorithms that use space that is sub-linear in the input size.

Max-k-Cover is a classic NP-Hard problem that has a wide range of applications including facility and sensor allocation [105], information retrieval [11], influence maximization in marketing strategy design [100], and the blog monitoring problem where we want to choose a small number of blogs that cover a wide range of topics [138]. It is well-known that the greedy algorithm, which greedily picks the set that covers the most number of uncovered elements, is a $e/(e-1)$ approximation and that unless $P = NP$, this approximation factor is the best possible [61].

Max-k-UniqueCover was first studied in the offline setting by Demaine et al. [49]. A motivating application for this problem was in the design of wireless networks where we want to place base stations that cover mobile clients. Each station could cover multiple clients but unless a client is covered by a unique station the client would experience too much interference. Demaine et al. [49] gave a polynomial time $O(\log k)$ approximation. Furthermore, they showed that Max-k-UniqueCover is hard to approximate within a factor $O(\log^{\sigma} n)$ for some constant σ under reasonable complexity assumptions. Erlebach and van Leeuwen [58] and Ito et al. [85] considered a geometric variant of the problem and Misra et al. [126] considered the parameterized complexity of the problem. This problem is also closely related to Minimum Membership Set Cover where one has to cover every element and minimizes the maximum overlap on any element [52, 106].

In the streaming set model, Max-k-Cover and the related SetCover problem[1] have both received a significant amount of attention [15, 34, 55, 78, 82, 83, 120, 138]. The most relevant result is a single-pass $2+\epsilon$ approximation using $\tilde{O}(k\epsilon^{-2})$ space [16, 120] although better approximation is possible in a similar amount of space if multiple passes are permitted [120]

[1] That is, find the minimum number of sets that cover the entire universe.

or if the stream is randomly ordered [129]. In this chapter, we almost exclusively consider single-pass algorithms where the sets arrive in an arbitrary order. The unique coverage problem has not been studied in the data stream model although it, and Max-k-Cover, are closely related to various graph problems that have been studied.

3.1.1 Our Results

Our main results are the following single-pass stream algorithms[2]:

(A) Bounded Set Cardinality. If all sets have size at most d, there exists a $\tilde{O}(d^{d+1}k^d)$ space data stream algorithm that solves Max-k-UniqueCover and Max-k-Cover exactly. We show that this is nearly optimal in the sense that any exact algorithm requires $\Omega(k^d)$ space.

(B) Bounded Multiplicity. If all elements occurs in at most r sets, we present the following algorithms:

- (B1) Max-k-UniqueCover: There exists a $2+\epsilon$ approximation algorithm using $\tilde{O}(\epsilon^{-3}k^2r)$ space.
- (B2) Max-k-UniqueCover: We show that the approximation factor can be improved to $1+\epsilon$ at the expense of increasing the space use to $\tilde{O}(\epsilon^{-4}k^3r)$.
- (B3) Max-k-Cover: There exists a $1+\epsilon$ approximation algorithm using $\tilde{O}(\epsilon^{-3}k^2r)$ space.

In contrast to the above results, when d and r are arbitrary, constant pass $1+\epsilon$ approximation algorithm for either problem requires $\Omega(\epsilon^{-2}m)$ space [14].[3] We also generalize of lower bound for Max-k-Cover [120] to Max-k-UniqueCover to show that any constant-pass algorithm with an approximation better than $e^{1-1/k}$ requires $\Omega(m/k^2)$ space. We also present a single-pass algorithm with an $O(\log \min(k, r))$ approximation for Max-k-UniqueCover using $\tilde{O}(k^2)$ space, i.e., the space is independent of r and d but the approximation factor depends on r. This algorithm is a simple combination of a Max-k-Cover algorithm due to [120] and an algorithm for Max-k-UniqueCover in the offline setting due to Demaine et al. [49]. Finally, our Max-k-Cover result (B3) algorithm also yields a new multi-pass result for SetCover. See Section 3.4.4 for details.

3.1.2 Technical Summary

Our results are essentially streamable kernelization results, i.e., the algorithm "prunes" the input (in the case of Max-k-UniqueCover and Max-k-Cover this corresponds to ignoring some of the input sets) to produce a "kernel" in such a way that a) solving the problem optimally on the kernel yields a solution that is as good (or almost as good) as the optimal solution on the original input and b) the kernel is streamable and sufficiently smaller than the original input such that it is possible to find an optimal solution for the kernel in significantly

[2]Throughout we use $\tilde{O}$ to denote that logarithmic factors of m and n are being omitted.

[3]The lower bound result by Assadi [14] was for the case of Max-k-Cover but we will explain that it also applies in the case of Max-k-UniqueCover.

less time than it would take to solve on the original input. In the field of fixed parameter tractability, the main requirement is that the kernel can be produced in polynomial time. In the growing body of work on streaming kernelization [37–39] the main requirement is that the kernel can then be constructed using small space in the data stream model. Our results fits in with this line of work and the analysis requires numerous combinatorial insights into the structure of the optimum solution for Max-k-UniqueCover and Max-k-Cover.

Our technical contributions can be outlined as follows.

- Results (A) and (B3) rely on various structural and combinatorial observations. At a high level, Result (A) uses the observation that each set of any Max-k-Cover or Max-k-UniqueCover solution intersects any maximal set of disjoint sets. The main technical step is to demonstrate that storing a small number of intersecting sets suffices to preserve the optimal solution.

- The $1+\epsilon$ and $2+\epsilon$ approximations for Max-k-Cover and Max-k-UniqueCover, i.e., results (B1) and (B3), are based on a very simple idea of first collecting the largest $O(rk/\epsilon)$ sets and then solving the problem optimally on these sets. This can be done in a space efficient manner using existing sketch for F_0 estimation in the case of Max-k-Cover and a new sketch we present the case of Max-k-UniqueCover. While the approach is simple, showing that it yields the required approximations requires some work and builds on a recent result by Manurangsi [111]. We also extend the algorithm to the model where sets can be inserted and deleted in a non-trivial way.

3.1.3 Comparison to related work.

In the context of streaming algorithms, for the Max-k-Cover problem, McGregor and Vu [119] showed that any approximation better than $1/(1-1/e)$ requires $\Omega(m/k^2)$ space. For the more general problem of streaming submodular maximization subject to a cardinality constraint, Feldman et al. [64] very recently showed a stronger lower bound that any approximation better than 2 requires $\Omega(m)$ space. Our results provide a route to circumvent these bounds via parameterization on k, r, and d.

Result (B3) leads to a parameterized algorithm for streaming SetCover. This new algorithm uses $\tilde{O}(rk^2n^\delta + n)$ space which improves upon the algorithm by Har-Peled et al. [78] that uses $\tilde{O}(mn^{1/\delta}+n)$ space, where k is an upper bound for the size of the minimum set cover, in the case $rk^2 \ll m$. Both algorithms use $O(1/\delta)$ passes and yield an $O(1/\delta)$ approximation.

In the context of offline parameterized algorithms, Bonnet et al. [25] showed that Max-k-Cover is fixed-parameter tractable in terms of k and d. However, their branching-search algorithm is not streamable. Misra et al. [126] showed that the maximum unique coverage problem in which the aim is to maximize the number of uniquely covered elements u without any restriction on the number of sets in the solution is fixed-parameter tractable. This problem admits a kernel of size 4^u. On the other hand, they showed that the budgeted version of this problem (where each element has a profit and each set has a cost and the goal is maximize the profit subject to a budget constraint) is $W[1]$-hard when parameterized by

the budget [4]. In this context, our result shows that a parameterization on both the maximum set size d and the budget k is possible (at least when all costs and profits are unit).

3.2 Preliminaries

3.2.1 Notation and Parameters

Throughout the chapter, m will denote the number of sets, n will denote the size of the universe, and k will denote the maximum number of sets that can be used in the solution. Given input sets $S_1, S_2, \ldots, S_m \subset [n]$, let

$$d = \max_i |S_i|$$

be the maximum set size and let

$$r = \max_j |\{i : j \in S_i\}|$$

be the maximum number of sets that contain the same element.

3.2.2 Structural Preliminaries

Given a collection of sets $C = \{S_1, S_2, \ldots, S_m\}$, we say a sub-collection $C' \subset C$ is a *matching* if the sets in C' are mutually disjoint. C' is a maximal matching if there does not exist $S \in C \setminus C'$ such that S is disjoint from all sets in C'. The following simple lemma will be useful at various points in the chapter.

Lemma 22. *For any input C, let $O \subset C$ be an optimal solution for either the* Max-k-Cover *or* Max-k-UniqueCover *problem. Let M_i be a maximal matching amongst the input set of size i. Then every set of size i in O of size intersects some set in M_i.*

Proof. Let $S \in O$ have size i. If it was disjoint from all sets in M_i then it could be added to M_i and the resulting collection would still be a matching. This violates the assumption that M_i is maximal. □

The next lemma extends the above result to show that we can potentially remove many sets from each M_i and still argue that there is an optimal solution for the original instance amongst the sets that intersect a set in some M_i.

Lemma 23. *Consider an input of sets of size at most d. For $i \in [d]$, let M_i be a maximal matching amongst the input set of size i and let M_i' be an arbitrary subset of M_i of size $\min(k + dk, |M_i|)$. Let D_i be the collection of all sets that intersect a set in M_i'. Then $\bigcup_i (D_i \cup M_i')$ contains an optimal solution to both the* Max-k-UniqueCover *and* Max-k-Cover *problem.*

[4] In the Max-k-UniqueCover problem that we consider, all costs and profits are one and the budget is k.

Proof. If $|M_i| = |M'_i|$ for all $1 \leq i \leq d$ then the result follows from Lemma 22. Suppose that If not, let $j = \max\{i \in [d] : |M_i| > |M'_i|\}$. Let O be an optimal solution and let O_i be all the sets in O of size i. We know that every set in $O_d \cup O_{d-1} \cup \ldots \cup O_{j+1}$ is in

$$\bigcup_{i \geq j+1} (D_i \cup M'_i) = \bigcup_{i \geq j+1} (D_i \cup M_i) .$$

Hence, the number of elements (uniquely) covered by O is at most the number of elements (uniquely) covered by $O_d \cup O_{d-1} \cup \ldots \cup O_{j+1}$ plus kj since every set in $O_j \cup \ldots \cup O_1$ (uniquely) covers at most j additional elements. But we can (uniquely) cover at least the number of elements (uniquely) covered by $O_d \cup O_{d-1} \cup \ldots \cup O_{j+1}$ plus kj. This is because M_j contains $k + dk$ disjoint sets of size j and at least $k + dk - kd = k$ of these are disjoint from all sets in $O_d \cup O_{d-1} \cup \ldots \cup O_{j+1}$. Hence, there is a solution amongst $\bigcup_{i \geq j}(D_i \cup M'_i)$ that is at least as good as O and hence is also optimal. □

3.2.3 Sketches and Subsampling

Coverage Sketch. Given a vector $x \in \mathbb{R}^n$, $F_0(x)$ is defined as the number of elements of x which are non-zero. If given a subset $S \subset \{1, \ldots, n\}$, we define $x_S \in \{0,1\}^n$ to be the characteristic vector of S (i.e., $x_i = 1$ iff $i \in S$) then given sets $S_1, S_2, \ldots$ note that $F_0(x_{S_1} + x_{S_2} + \ldots)$ is exactly the number of elements covered by $S_1 \cup S_2 \cup \ldots$. We will use the following result for estimating F_0.

Theorem 24 ([17, 45]). *There exists an $\tilde{O}(\epsilon^{-2} \log \delta^{-1})$-space algorithm that, given a set $S \subseteq [n]$, can construct a data structure $\mathcal{M}(S)$, called an F_0* sketch *of S, that has the property that the number of distinct elements in a collection of sets $S_1, S_2, \ldots, S_t$ can be approximated up to a $1 + \epsilon$ factor with probability at least $1 - \delta$ given the collection of F_0 sketches $\mathcal{M}(S_1), \mathcal{M}(S_2), \ldots, \mathcal{M}(S_t)$.*

Note that if we set $\delta \ll 1/(\text{poly}(m) \cdot \binom{t}{k})$ in the above result we can try collection of k sets amongst $S_1, S_2, \ldots, S_t$ and get a $1 + \epsilon$ approximation for the coverage of each collection with high probability.

The Subsampling Framework. Assuming we have v such that $\text{OPT}/2 \leq v \leq \text{OPT}$. Let $h : [n] \to \{0,1\}$ be a hash function that is $\Omega(\epsilon^{-2} k \log m)$-wise independent. We run our algorithm on the subsampled universe $U' = \{u \in U : h(u) = 1\}$. Furthermore, let

$$\mathbb{P}[h(u) = 1] = p = \frac{ck \log m}{\epsilon^2 v}$$

where c is some sufficiently large constant. Let $S' = S \cap U'$ and let OPT' be the optimal unique coverage value in the subsampled set system. The following result is from McGregor and Vuj [120]. We note that the proof is the same except that the indicator variables now correspond to the events that an element being uniquely covered (instead of being covered).

Lemma 25. *With probability at least* $1 - 1/\operatorname{poly}(m)$*, we have that*

$$p \operatorname{OPT}(1+\epsilon) \geq \operatorname{OPT}' \geq p \operatorname{OPT}(1-\epsilon)$$

Furthermore, if $S_1, \ldots, S_k$ *satisfies* $UC(\{S'_1, \ldots, S'_k\}) \geq p \operatorname{OPT}(1-\epsilon)/t$ *then*

$$UC(\{S_1, \ldots, S_k\}) \geq \operatorname{OPT}(1/t - 2\epsilon) .$$

We could guess $v = 1, 2, 4, \ldots, n$. One of the guesses must be between $\operatorname{OPT}/2$ and OPT which means $\operatorname{OPT}' = O(\epsilon^{-2} k \log m)$. Furthermore, if we find a $1/t$ approximation on the subsampled universe, then that corresponds to a $1/t - 2\epsilon$ approximation in the original universe. We note that as long as $v \leq \operatorname{OPT}$ and h is $\Omega(\epsilon^{-2} k \log m)$-wise independent, we have (see [141], Theorem 5):

$$\begin{aligned}
&\mathbb{P}\left[UC(\{S'_1, \ldots, S'_\ell\}) = p \cdot UC(\{S_1, \ldots, S_\ell\}) \pm \epsilon p \operatorname{OPT}\right] \\
\geq &1 - \exp\left(-\Omega(k \log m)\right) \\
\geq &1 - 1/m^{\Omega(k)} .
\end{aligned}$$

This gives us Lemma 25 even for when $v < \operatorname{OPT}/2$. However, if $v \leq \operatorname{OPT}/2$, then OPT' may be larger than $O(\epsilon^{-2} k \log m)$, and we may use too much memory. To this end, we simply terminate those instantiations. Among the instantiations that are not terminated, we return the solution given by the smallest guess.

Unique Coverage Sketch. For unique coverage, our sketch of a set corresponds to subsampling the universe via some hash function $h : [n] \to \{0, 1\}$ where h is chosen randomly such that for each i, $\mathbb{P}[h(i) = 1] = p$ for some appropriate value p. Specifically, rather processing an input set S, we process $S' = \{i \in S : h(i) = 1\}$. Note that $|S'|$ has size $p|S|$ in expectation. This approach was use by McGregor and Vu [120] in the context of Max-k-Cover and extends easily to Max-k-UniqueCover via the subsampling approach in 3.2.3. The consequence is that if there is a streaming algorithm that finds a t approximation, we can turn that algorithm into a $t(1+\epsilon)$ approximation algorithm in which we can assume that $\operatorname{OPT} = O(\epsilon^{-2} k \log m)$ with high probability[5] by running the algorithm on a subsampled sets rather than the original sets. Note that this also allows us to assume input sets have size $O(\epsilon^{-2} k \log m)$ since $|S'| \leq \operatorname{OPT}$. Hence each "sketched" set can be stored in $B = O(\epsilon^{-2} k \log m \log n)$ bits.

Algorithm with $\Omega(m)$ Memory. We will use the above sketches in a more interesting context later in the chapter, note that they immediately imply a trivial algorithmic result. Consider the naive algorithm that stores every set and finds the best solution; note that this requires exponential time. We note that since we can assume $\operatorname{OPT} = O(\epsilon^{-2} k \log m)$, each set has size at most $O(\epsilon^{-2} k \log m)$. Hence, we need $\tilde{O}(\epsilon^{-2} m k)$ memory to store all the sets. This approach was noted in [120] in the context of Max-k-Cover but also apples to Max-k-UniqueCover. We will later explain that for a $1+\epsilon$ approximation, the above trivial algorithm is optimal up to polylogarithmic factors for constant k.

[5]Throughout this chapter, we say an algorithm is correct with high probability if the probability of failure is inversely polynomial in m.

3.3 Exact Algorithms

Let C be the input sets. In this section we will initially assume all input sets have size exactly d and will show that there exists a single-pass data stream algorithm that uses $\tilde{O}(d^{d+1}k^d)$ space and returns a collection of sets $C' \subset C$ such that the optimal solution for either the maximum coverage or unique coverage problem when restricted to C' is equal to the optimal solution with no such restriction. We will subsequently generalize this to the case when sets can have any size at most d. In this section we will assume that r can be unbounded, e.g., an element in the universe could appear in m of the input sets.

3.3.1 Warm-Up Idea

Appealing to Lemma 22, we know that sets in an optimal solution to maximum coverage or maximum unique coverage intersect with a maximal matching. Hence, a natural approach is to construct a maximimal matching A greedily as the sets arrive along with any set that intersects a set in A. If the maximal matching ever exceeds size k then we have an optimal solution to Max-k-Cover and Max-k-UniqueCover that covers dk elements and hence we can ensure $|A| \leq k$. However, a set in A could intersect with $\Omega(m)$ other sets in the worst case[6] The main technical step in the algorithm in the next section is a way to carefully store only some of the sets that intersect A such that we can bound the number of stored sets in terms of k and d and yet still assume that stored sets include an optimal solution to either Max-k-Cover or Max-k-UniqueCover.

3.3.2 Algorithm

(1) Let A and X_u (for all $u \in [n]$) be empty sets. Each will correspond to a collection of sets. Let $b = d(k-1)$.

(2) Process the stream and let S be the next set:

   (a) If S is disjoint from all sets in A and $|A| < k$, add S to A.

   (b) If $u \in S \cap S'$ for some $S' \in A$:

      i. Add S to X_u if there does not exist a subset $T \subset (S \setminus \{u\})$ that occurs as subset of $(b+1)^{d-1-|T|}$ sets in X_u.

(3) Return the best solution in $C' = A \cup (\bigcup_u X_u)$.

3.3.3 Analysis

We start with the following combinatorial lemma[7].

[6] It can be bounded in terms of d and r however. Specifically, each set can intersect with at most $d(r-1)$ other sets. However, in this section we are assuming r is unbounded so this bound does not help us here.

[7] For the interested reader who is familiar with the relevant combinatorial results, we note that we can prove a similar lemma to the one here via the Sunflower Lemma [10, 134]. In particular, one can argue that there exists a sufficiently large sunflower amongst $\{S \in X : T^* \text{ is a subset of } S\}$ whose core includes T^*. With some small adjustment to the subsequent theorem, this would be sufficient for our purposes. However, we instead include this version of the lemma because it is simpler and self-contained.

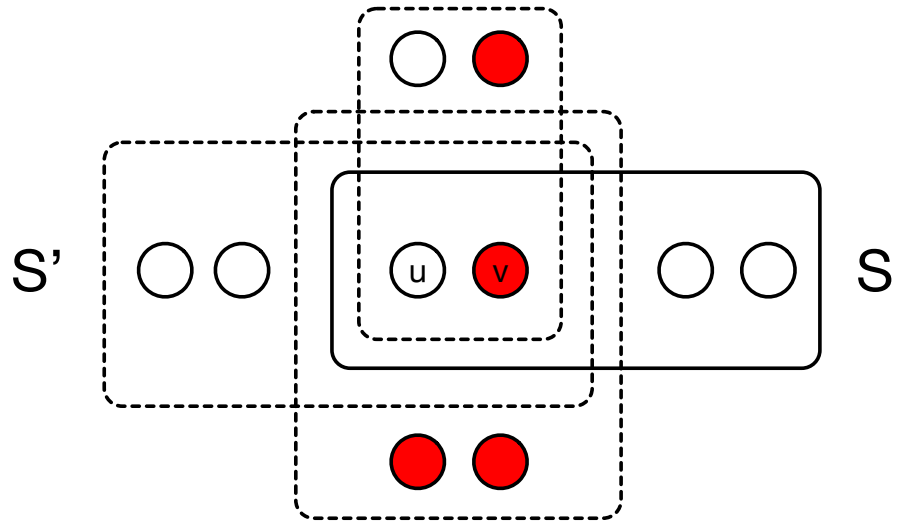


Figure 3.1: An example where all sets have size 4. Suppose the three dotted sets are currently stored in X_u. If S intersects u, it may not be added to X_u even if S is in an optimal solution O. In the above diagram, the elements covered by sets in $O \setminus \{S\}$ are shaded (note that the sets in O other than S are not drawn). In particular, if a subset T of $S \setminus \{u\}$ is a subset of many sets currently stored in X_u, it will not be added. For example, $T = \{v\}$ already occurs in the three subsets currently in X_u and, for the sake of a simple diagram, suppose 3 is the threshold for the maximum number of times a subset may appear in sets in X_u. Our analysis shows that there always exists a set S' in X_u that is "as good as" S in the sense that $S' \cap S = T \cup \{u\}$ and all the elements in $S' \setminus S$ are elements not covered by sets in $O \setminus \{S\}$.

Lemma 26. *Let $X = \{S_1, S_2, \ldots\}$ be a collection of distinct sets where each $S_i \subset [n]$ and $|S_i| = a$. Suppose for all $T \subseteq [n]$ with $|T| \leq a$ there exists at most*

$$\ell_{|T|} := (b+1)^{a-|T|}$$

sets in X that contain T. Furthermore suppose there exists a set T^* such that this inequality is tight. Then, for all $B \subset [n]$ disjoint from T^* with $|B| \leq b$ there exists $S_i \in X$ such that $T^* \subset S_i$ and $|S_i \cap B| = 0$.

Proof. If $|T^*| = a$ then $T^* \in X$ and this set satisfies the necessary conditions. Henceforth, assume $|T^*| < a$. Consider the $\ell_{|T^*|}$ sets in X that are supersets of T^*. Call this collection X'. For any $x \in B$, there are at most $\ell_{|T^*|+1}$ sets that include $T^* \cup \{x\}$. Since there are b choices for x, at most

$$b\ell_{|T^*|+1} = b(b+1)^{a-|T^*|-1} < (b+1)^{a-|T^*|} = \ell_{|T^*|}$$

sets in X' contains an element in B. Hence, at least one set in X does not contain any element in B. □

For any collection of sets F, let $f(F)$ be the maximum coverage of at most k sets in F and let $g(F)$ be the maximum unique coverage of at most k set in F.

Theorem 27. *The output of the algorithm satisfies $f(C') = f(C)$ and $g(C') = g(C)$.*

Proof. Let C_0 be the union of A and all sets that intersect a set in A, i.e.,

$$C_0 = \{S \in A\} \cup \{S \in C : |S \cap S'| > 0 \text{ for some } S' \in A\} .$$

Note that every set in the optimum solution of maximum coverage intersects with some set in A and hence $f(C_0) = f(C)$. For $i \geq 1$ consider,

$$C_i = C_0 \setminus \{\text{first } i \text{ sets in stream that are not in output } C'\} .$$

We will next argue that for any $i \geq 0$, $f(C_{i+1}) = f(C_i)$ and the theorem follows.

Let O be an optimum solution in C_i and let $\{S\} = C_i \setminus C_{i+1}$. If $S \notin O$ then clearly $f(C_{i+1}) = f(C_i)$ since $O \subseteq C_{i+1}$. If $S \in O$ but not in C_{i+1} then let $u \in S$ be the node for which we contemplated adding S to X_u but didn't because of the additional requirements.

Claim 2. *There exists S' in X_u such that $f((O \setminus \{S\}) \cup \{S'\}) = f(C_i)$ as required.*

Proof of Claim. If S was not added to X_u there exists a subset of $T^* \subset (S \setminus \{u\})$ that is a subset of $(b+1)^{d-1-|T^*|}$ sets in X_u. Let X be the collection of sets of size $a = d-1$ formed by removing u from each of the sets in X_u. Note that X satisfies the assumptions of Lemma 26. Let B be the set of at most $b = d(k-1)$ elements in the set

$$B = \{v : v \in S'' \text{ for some } S'' \in O\} \setminus S .$$

By Lemma 26, there exists a set S' in X such that $T^* \subset S'$ and $|(S' \setminus T^*) \cap B| = 0$. Hence, $f((O \setminus \{S\}) \cup \{S'\})$. □

The proof for unique coverage, i.e., $g()$, is identical. □

Lemma 28. *The space used by the algorithm is $\tilde{O}(d^{d+1}k^d)$.*

Proof. Recall that one of the requirements for a set S to be added to X_u is that the number of sets in X_u that are supersets of any subset of $S \setminus \{u\}$ of size t is at most $(b+1)^{d-1-t}$. This includes the empty subset and since every set in X_u is a superset of the empty set, we deduce that

$$|X_u| \leq (b+1)^{d-1-0} = (b+1)^{d-1}.$$

Since $|A| \leq k$, the number of sets that are stored is at most

$$\begin{aligned} |A| + \sum_{u \in \cup_{S \in A}} |X_u| &\leq |A| + d|A| \cdot (b+1)^{d-1} \\ &\leq |A| + d|A| \cdot O((dk)^{d-1}) \\ &= O((dk)^d) . \end{aligned}$$

□

3.3.4 Generalization to Sets of Different Size

In the case where sets may have any size at most d, we run the algorithm described in Section 3.3.2 in parallel for stream sets of each size $t \in [d]$. By appealing to Lemma 23, we know that an optimal maximum overage or maximum unique coverage intersects with the union of maximal matchings of sets of size t for each t. We again rely on Lemma 26 to establish that we can store only some of the sets that intersect these matchings and still retain an optimal solution to either coverage problem. We describe the algorithm below.

(1) Let A_t and $X_{u,t}$ (for all $u \in [n]$ and $t \in [d]$) be empty sets. Each will correspond to a collection of sets. Let $b = d(k-1)$.

(2) Process the stream and let S be the next set and let $t = |S|$:

   (a) If S is disjoint from all sets in A_t and

$$|A_t| < \begin{cases} dk + k & \text{if } t < d \\ k & \text{if } t = d \end{cases}$$

   then add S to A_t.

   (b) If $u \in S \cap S'$ for some $S' \in A_t$:

      i. Add S to $X_{u,t}$ if there does not exist a subset $T \subset (S \setminus \{u\})$ that occurs as subset of $(b+1)^{t-1-|T|}$ sets in X_u.

(3) Return $C'' = (\bigcup_t A_t) \cup \left(\bigcup_{u,t} X_{u,t}\right)$.

Theorem 29. *The output of the algorithm satisfies $f(C'') = f(C)$ and $g(C'') = g(C)$.*

Proof. Let

$$C_0 = \bigcup_t \left(\{S \in A_t\} \cup \{S \in C : |S \cap S'| > 0 \text{ for some } S' \in A\}\right).$$

Define C_i, O, and S as in the proof of Theorem 27. By Lemma 23, $f(C_0) = f(C)$ since there is an optimum solution of maximum coverage in which every set intersects with some set A_t. Let $u \in S$ be the node which prevented us from adding S to $X_{u,t}$. We now prove an analog of Claim 2 which implies that for any $i \geq 0$, $f(C_{i+1}) = f(C_i)$.

Claim 3. *There exists S' in $X_{u,t}$ such that $f((O \setminus \{S\}) \cup \{S'\}) = f(C_i)$ as required.*

Proof of Claim. If S was not added to $X_{u,t}$ there exists of a subset of $T^* \subset (S \setminus \{u\})$ that is a subset of $(b+1)^{t-1-|T^*|}$ sets in X_u. Let X be the collection of sets of size $a = t-1$ formed by removing u from each of the sets in X_u. X satisfies the assumptions of Lemma 26. Let B be the set of at most $b = d(k-1)$ elements in the set

$$B = \{v : v \in S'' \text{ for some } S'' \in O\} \setminus S\,.$$

By Lemma 26, there exists a set S' in X such that $T^* \subset S'$ and $|(S' \setminus T^*) \cap B| = 0$. Hence, $f((O \setminus \{S\}) \cup \{S'\})$ □

Again, the proof is identical for unique coverage. □

Lemma 30. *The space used by the algorithm is* $\tilde{O}(d^{d+1}k^d)$.

Proof. For all t, $|X_{u,t}| \leq (b+1)^{t-1}$. Since $|A_d| \leq k$ and $|A_t| = O(dk)$ for $t < k$, the number of sets stored is at most:

$$\begin{aligned}
&\sum_{t=1}^{d} \left(|A_t| + \sum_{u \in \bigcup_{S \in A_t}} |X_{u,t}| \right) \\
&\leq O(d^2k + d^2k(1 + (b+1) + \ldots + (b+1)^{d-2}) + dk(b+1)^{d-1}) \\
&= O((dk)^d) \ .
\end{aligned}$$

□

We summarize the result as a theorem.

Theorem 31. *There exists a single-pass,* $\tilde{O}(d^{d+1}k^d)$*-space algorithm that yields an exact solution to* Max-k-Cover *and* Max-k-UniqueCover.

3.3.5 An Algorithm for Insert/Delete Streams

Chitnis et. al. [38] introduce a sketching primitive $\mathsf{Sample}_{\gamma,d}$ suitable for insert/delete data streams which is capable of randomly sampling a diverse selection of sets. $\mathsf{Sample}_{\gamma,d}$ first assigns a color to each element from γ colors uniformly at random using a d-wise independent hash function. Each set in C is therefore associated with the multiset of colors assigned to its elements (a "color signature"). By maintaining an ℓ_0-sampler for all sets of each color signature, It is possible to sample one set of each color signature from $\mathsf{Sample}_{\gamma,d}$ at the end of stream. The following lemma establishes that these sampled sets are likely to include optimal solutions for maximum coverage and maximum unique coverage.

Lemma 32. *Let* C' *be the collection of sets sampled from* $\mathsf{Sample}_{4k^2d^2,d}$*. Then* $\mathbb{P}[f('C) = f(C)] \geq 3/4$ *and* $\mathbb{P}[g(C') = g(C)] \geq 3/4$.

Proof. Let M be an optimal solution to Max-k-Cover and let $c : U \to [\gamma]$ be the coloring of sets defined by $\mathsf{Sample}_{4k^2d^2,d}$. Order the k sets in M arbitrarily and let m_i be the ith set in this ordering. We will argue that with good probability, the ith set's color signature has no colors in common with any earlier set in the ordering, except for those of elements which are already covered. More formally, let $col(S) := \{w | c(e) = w | e \in S\}$ and let H_i denote the event $|col(m_i) \cap col(\bigcup_{j<i} m_j)| = |m_i \cap (\bigcup_{j<i} m_j)|$. Note that if H_i occurs then $\mathsf{Sample}_{4k^2d^2,d}$ either returns m_i or some other set with the same color signature which can replace m_i in the optimal solution (since its color signature guarantees its intersection with

the rest of the optimal solution is not greater than m_i's). Since our hash function is d-wise independent:

$$\mathbb{P}[H_i] \geq \left(1 - \frac{i \cdot d}{\gamma}\right)^d = \left(1 - \frac{i}{4k^2 d}\right)^d \approx e^{-i/4k^2}.$$

The above inequality holds because the earlier i edges have at most $i \cdot d$ unique colors. If H_i occurs for all $0 \leq i \leq k-1$,

$$\prod_{i=0}^{k-1} \mathbb{P}[H_i] \geq \prod_{i=0}^{k-1} e^{-i/4k^2} = \exp\left(-\sum_{i=0}^{k-1}(i/4k^2)\right)$$
$$= \exp\left(-\frac{(k-1)(k-2)}{4k^2}\right) \geq e^{-1/4} \geq 3/4.$$

□

A $\mathsf{Sample}_{4k^2d^2,d}$ sketch maintains an ℓ_0 sketch for each of γ^d color signatures, requiring $tO((kd)^{2d})$ space. Additionally, the d-wise independent hash function mapping nodes to $4k^2d^2$ colors uses $O(d \log(4k^2d^2) = \tilde{O}(d)$ space, so the entire $\mathsf{Sample}_{4k^2d^2,d}$ sketch requires $tO((kd)^{2d})$ space. Constructing $log(k)$ $\mathsf{Sample}_{4k^2d^2,d}$ sketches in parallel[8] guarantees that $f('C) = f(C)$ and $g('C) = g(C)$ with probability $1 - 1/\operatorname{poly}(k)$. This gives the following theorem.

Theorem 33. *There exist randomized single-pass algorithms using $\tilde{O}((kd)^{2d})$ space and allowing deletions that yield an exact solution to* Max-k-Cover *and* Max-k-UniqueCover.

3.4 Approximation Algorithms

In this section, we present a variety of different approximation algorithms where the space used by the algorithm is independent of d but, in some cases, may depend on r.

3.4.1 Unique Coverage: $2+\epsilon$ Approximation

In this section, we present a $2+\epsilon$ approximation for unique coverage. The algorithm is simple but the analysis is non-trivial. The algorithm stores the ηk largest sets where $\eta = \lceil r/\epsilon \rceil$ and finds the best unique coverage achievable by selecting at most k of these sets.

We will present an algorithm with a $1+\epsilon$ approximation in the next subsection with the expense of an extra k/ϵ factor in the space use. However, the algorithm in this section is appealing in the sense that it is much simpler and can be extended to insertion-deletion streams. The analysis of this approach may also be of independent interest.

Let C' be the ηk sets of largest size. To find the best solution C'' amongst C', we use the unique coverage sketches presented in the Section 3.2. Note that to find the ηk largest sets we just store the sizes of sets sketched so far along with their unique coverage sketches. Finally, we return the best solution C'' using most k sets in C' based on the unique coverage sketches that we store. Recall that each unique coverage sketch requires $\tilde{O}(k/\epsilon^2)$ space. We have the following result.

[8]Note that each $\mathsf{Sample}_{4k^2d^2,d}$ sketch has a different random coloring.

Theorem 34. *There exists a randomized single-pass algorithm using $\tilde{O}(\epsilon^{-2}\eta k) = \tilde{O}(\epsilon^{-3}k^2 r)$ space algorithm that $2+\epsilon$ approximates* Max-k-UniqueCover.

Proof. Let the sizes of the ηk largest sets be (with arbitrarily tie-breaking) be $d_1 \geq d_2 \geq \ldots \geq d_{\eta k}$ and let

$$d^* := \frac{d_1 + \ldots + d_k}{k} \quad \text{and} \quad d' := \frac{d_{k+1} + \ldots + d_{\eta k}}{(\eta - 1)k} .$$

Let $\mathcal{O}$ be an optimal collection of sets for Max-k-UniqueCover. First, we observe that for each set $S \in \mathcal{O} \setminus C'$, we have that $|S| \leq d_{\eta k} \leq d'$. Hence,

$$\mathrm{OPT} \leq f(\mathcal{O} \cap C') + \sum_{S \in \mathcal{O} \setminus C'} |S| \leq h(C'') + kd' .$$

where $h()$ is a function of a collection of sets that returns the number of elements that are covered by exactly one of these sets. Thus, if $kd' < 0.5\,\mathrm{OPT}$, then it is immediate that the number of elements uniquely covered by our solution is $h(C'') > 0.5\,\mathrm{OPT}$.

Now we consider the case $kd' \geq 0.5\,\mathrm{OPT}$. For the sake of analysis, consider randomly partitioning C' into a set C'_1 of size k and $C'_2 = C' \setminus C'_1$. Observe that

$$\begin{aligned}
&\mathbb{E}\left[h(C'_1)\right] \\
&= \sum_{S \in C'} \mathbb{E}\left[\#\text{ of elements uniquely covered by } S \text{ in } C'_1\right] \\
&= \sum_{S \in C'} \sum_{u \in S} \mathbb{P}\left[S \in C'_1 \text{ and } u \text{ is uniquely covered in } C'_1\right] \\
&\geq \sum_{S \in C'} \sum_{u \in S} \left(\mathbb{P}\left[S \in C'_1\right] - \sum_{S' \in C \setminus \{S\} : u \in S'} \mathbb{P}\left[S \in C'_1, S' \in C'_1\right] \right) \\
&\geq \sum_{S \in C'} |S| \left(\epsilon/r - (r-1)(\epsilon/r)^2\right) \\
&\geq \eta k \cdot d' \left(\epsilon/r - r(\epsilon/r)^2\right) \geq kd'\,(1-\epsilon) \geq (1-\epsilon)\,\mathrm{OPT}\,/2 .
\end{aligned}$$

□

We note that it is possible to improve the result slightly by setting $\eta = \sqrt{2/\epsilon}$ for the bound of for $\mathbb{E}\left[h(C'_1)\right]$ in the proof of Theorem 34 as follows:

$$
\begin{aligned}
\mathbb{E}\left[h(C_1')\right] &= \sum_{S \in C'} \mathbb{E}\left[\text{\# of elements uniquely covered by } S \text{ in } C_1'\right] \\
&\geq \sum_{S \in C'} \sum_{i \in S} \frac{k}{\eta k} \frac{(\eta-1)k}{\eta k - 1} \\
&= \frac{1}{\eta} \cdot \left(\frac{(\eta-1)k}{\eta k - 1}\right) \sum_{S \in C'} |S| \\
&> \frac{\eta - 1}{\eta^2} \left(k d^* + (\eta - 1) k d'\right) \\
&\geq \mathrm{OPT}(1/\eta - 1/\eta^2 + (1 - 1/\eta)^2 0.5) \\
&= \mathrm{OPT}(0.5 - 0.5/\eta^2) \\
&= (0.5 - \epsilon)\,\mathrm{OPT}\ .
\end{aligned}
$$

The space used in resulting algorithm scales with $\epsilon^{-2.5}$ rather than ϵ^{-3} as implied by the analysis for general r.

Extension to Insert/Delete Streams

We now explain how the above approach can be extended to the case where sets may be inserted and deleted. In this setting, it is not immediately obvious how to select the largest ηk sets; the approach used when sets are only inserted does not extend. Note that in this model we can set m to be that maximum number of sets that have been inserted and not deleted at any prefix of the stream rather than the total number of sets inserted/deleted.

However, we can extend the result as follows. Suppose the sketch of a set for approximating maximum unique coverage requires B bits; recall from Section 3.2.3 that $B = k\epsilon^{-2}\,\mathrm{polylog}(n, m)$ suffices. We can encode such a sketch of a set S as an integer $i(S) \in [2^B]$. Suppose we know that exactly ηk sets have size at least some threshold t. We will remove this assumption shortly. Consider the vector $x \in [N]$ where $N = 2^B$ that is initially 0 and then is updated by a stream of set insertions/deletions as follows:

(1) When S is inserted, if $|S| \geq t$, then $x_{i(S)} \leftarrow x_{i(S)} + 1$.

(2) When S is deleted, if $|S| \geq t$, then $x_{i(S)} \leftarrow x_{i(S)} - 1$.

At the end of this process $x \in \{0, 1, \ldots, , m\}^{2^B}$, $\ell_1(x) = \eta k$, and reconstruct the sketches of largest ηk sets given x. Unfortunately, storing x explicitly in small space is not possible since, while we are promised that at the end of the stream $\ell_1(x) = \eta k$, during the stream it could be that x is an arbitrary binary string with m one's and this requires $\Omega(m)$ memory to store. To get around this, it is sufficient to maintain a linear sketch of x itself that support sparse recovery. For our purposes, the CountMin Sketch [46] is sufficient although other approaches are possible. The CountMin Sketch allows x to reconstructed probability $1 - \delta$ using a sketch of size

$$
O(\log N + \eta k \log(\eta k/\delta) \log m) = O(\eta k \epsilon^{-2}\,\mathrm{polylog}(n, m))\ .
$$

To remove the assumption that we do not know t in advance, we consider values:

$$
t_0, t_1, \ldots, t_{\lceil \log_{1+\epsilon} m \rceil} \text{ where } t_i = (1+\epsilon)^i\ .
$$

We define vector $x^0, x^1, \ldots \in \{0, 1, \ldots, , m\}^{2^B}$ where x^i is only updated when a set of size $\leq t_i$ but $> t_{i-1}$ is inserted/deleted. Then there exists i such that $\leq \eta k$ sets have size $\leq t_{i-1}$ and the sketches of these sets can be reconstructed from $x^0, \ldots, x^{t_{i-1}}$. To ensure we have ηk sets, we may need some additional sketches corresponding to sets of size $> t_{i-1}$ and $\leq t_i$ but unfortunately there could be m such sets and we are only guaranteed recover of x^{t_i} when it is sparse. However, if this is indeed the case we can still recover enough entries of x^{t_1} by first subsampling the entries at the appropriate rate (we can guess sampling rate $1, 1/2, 1/2^2, \ldots 1/m$) in the standard way. Note that we can keep track of $\ell_1(x^i)$ exactly for each i using $O(\log m)$ space.

The next section presents algorithms for maximum coverage and set cover using a similar approach based on keeping a set of the largest elements seen so far.

3.4.2 Maximum Coverage and Set Cover

In this section, we generalize the approach of Manurangsi [111] and combine that with F_0-sketch to obtain a $1+\epsilon$ approximation using $\tilde{O}(\epsilon^{-3}k^2r)$ space for the maximum coverage problem.

Manurangsi [111] showed that for the maximum k-vertex cover problem, the $\Theta(k/\epsilon)$ vertices with highest degrees form a $1+\epsilon$ approximation kernel. That is, there exist k vertices among those that cover $(1-\epsilon)\,\mathrm{OPT}$ edges. We now consider a set system in which an element belongs to at most r sets (this can also be viewed as a hypergraph where each set corresponds to a vertex and each element corresponds to a hyperedge; we then want to find k vertices that touch as many hyperedges as possible).

We begin with the following lemma that generalizes the aforementioned result in [111]. We may assune that $m \gg Crk/\epsilon$ for some large constant C; otherwise, we can store all the sets.

Lemma 35. *Suppose $m > \lceil rk/\epsilon \rceil$. Let K be the collection of $\lceil rk/\epsilon \rceil$ sets with largest sizes (tie-broken arbitrarily). There exist k sets in K that cover $(1-\epsilon)\,\mathrm{OPT}$ elements.*

Proof. Let $\mathcal{O}$ denote the collection of k sets in some optimal solution. Let $\mathcal{O}^{in} = \mathcal{O} \cap K$ and $\mathcal{O}^{out} = \mathcal{O} \setminus K$. We consider a random subset $Z \subset K$ of size $|\mathcal{O}^{out}|$. We will show that the sets in $Z \cup \mathcal{O}^{in}$ cover $(1-\epsilon)\,\mathrm{OPT}$ elements in expectation; this implies the claim.

Let $\chi(Z)$ denote the set of elements covered by the sets in Z. Let $[\mathcal{E}]$ denote the indicator variable for event $\mathcal{E}$. We rewrite

$$|\chi(Z \cup \mathcal{O}^{in})| = |\chi(\mathcal{O}^{in})| + |\chi(Z)| - |\chi(\mathcal{O}^{in}) \cap \chi(Z)|\,.$$

Furthermore, the probability that we pick a set S in K to add to Z is

$$p := \frac{|\mathcal{O}^{out}|}{|K|} \leq \frac{k}{kr/\epsilon} = \frac{\epsilon}{r}\,.$$

Next, we upper bound $\mathbb{E}\left[|\chi(\mathcal{O}^{in}) \cap \chi(Z)|\right]$. We have

$$\begin{aligned}\mathbb{E}\left[|\chi(\mathcal{O}^{in}) \cap \chi(Z)|\right] &\leq \sum_{u \in \chi(\mathcal{O}^{in})} \sum_{S \in K : u \in S} \mathbb{P}\left[S \in Z\right] \\ &\leq \sum_{u \in \chi(\mathcal{O}^{in})} rp \leq |\chi(\mathcal{O}^{in})| \cdot \epsilon\,.\end{aligned}$$

We lower bound $\mathbb{E}\left[|\chi(Z)|\right]$ as follows.

$$\begin{aligned}
&\mathbb{E}\left[|\chi(Z)|\right] \\
&\geq \mathbb{E}\left[\sum_{S\in K}\left(|S|[S\in Z] - \sum_{S'\in K\setminus\{S\}}|S\cap S'|[S\in Z\wedge S'\in Z]\right)\right] \\
&\geq \sum_{S\in K}\left(|S|p - \sum_{S'\in K\setminus\{S\}}|S\cap S'|p^2\right) \\
&\geq \sum_{S\in K}\left(|S|p - r|S|p^2\right) \geq p(1-pr)\sum_{S\in K}|S| \geq p(1-\epsilon)\sum_{S\in K}|S| \ .
\end{aligned}$$

In the above derivation, the second inequality follows from the observation that $\mathbb{P}\left[S\in Z\wedge S'\in Z\right]\leq p^2$. The third inequality is because $\sum_{S'\in K\setminus\{S\}}|S\cap S'|\leq r|S|$ since each element belongs to at most r sets.

For all $S\in K$, we must have have

$$|S| \geq \frac{\sum_{Y\in\mathcal{O}^{out}}|Y|}{|\mathcal{O}^{out}|} \geq \frac{|\chi(\mathcal{O}^{out})|}{|\mathcal{O}^{out}|} \ .$$

Thus,

$$\begin{aligned}
\mathbb{E}\left[|\chi(Z)|\right] \geq p\,(1-\epsilon)\,|K|\frac{|\chi(\mathcal{O}^{out})|}{|\mathcal{O}^{out}|} &= p\,(1-\epsilon)\,\frac{|\chi(\mathcal{O}^{out})|}{p} \\
&= (1-\epsilon)|\chi(\mathcal{O}^{out})| \ .
\end{aligned}$$

Putting it together,

$$\begin{aligned}
\mathbb{E}\left[|\chi(Z\cup\mathcal{O}^{in})|\right] &\geq |\chi(\mathcal{O}^{in})| + (1-\epsilon)|\chi(\mathcal{O}^{out})| - |\chi(\mathcal{O}^{in})|\cdot\epsilon \\
&\geq (1-\epsilon)\,\mathrm{OPT} \ .
\end{aligned}$$

□

With the above lemma in mind, the following algorithm's correctness is immediate.

(1) Store F_0-sketches of the kr/ϵ largest sets, where the failure probability of the sketches is set to $\frac{1}{\mathrm{poly}(n)\binom{m}{k}}$.

(2) At the end of the stream, return the k sets with the largest coverage based on the estimates given by the F_0-sketches.

We restate our result as a theorem.

Theorem 36. *There exists a randomized one-pass, $\tilde{O}(k^2r/\epsilon^3)$-space, algorithm that with high probability finds a $1+\epsilon$ approximation to* Max-k-Cover.

3.4.3 Unique Coverage: $1+\epsilon$ Approximation

The approximation factor in the previous section can be improved to $1+\epsilon$ at the expense of an extra factor of k/ϵ in the space. Recall in Section 3.3.1 that there exists an algorithm for solving Max-k-UniqueCover exactly by storing $O(kdr)$ sets, i.e., with $\tilde{O}(kd^2r)$ space. Combining this with the Subsampling Framework discussed in Section 3.2.3, we may assume $d \leq \mathrm{OPT} = O(\epsilon^{-2}k\log m)$. This immediately implies the following theorem.

Theorem 37. *There exists a randomized one-pass algorithm using $\tilde{O}(\epsilon^{-4}k^2r)$ space that finds a $1+\epsilon$ approximation of* Max-k-UniqueCover.

Note that the same approach would work for Max-k-Cover but we present a better result in Section 3.4.2.

3.4.4 Unique Coverage: $O(\log\min(k,r))$ Approx.

We now present an algorithm whose space does not depend on r but the result comes at the cost of increasing the approximation factor to $O(\log(\min(k,r)))$. It also has the feature that the running time is polynomial in k in addition to being polynomial in m and n.

The basic idea is as follows: We consider an existing algorithm that first finds a 2 approximation for the Max-k-Cover problem. Let the corresponding solution be C'. The algorithm then finds the best solution of Max-k-UniqueCover among the sets in C'.

Let z^* be a guess such that $(1-\epsilon)\,\mathrm{OPT}^* \leq z^* \leq \mathrm{OPT}^*$ where OPT^* is the value of the optimal Max-k-Cover.

(1) Initialize $T = \varnothing$ which will store sets from the stream.

(2) For each set S in the stream, if $|T| < k$ and

$$|(\cup_{A\in T}A)\cup S| - |\cup_{A\in T}A| \geq z^*/(2k)\ ,$$

then add S to T and store S in the memory.

(3) Return the best solution Q (in terms of unique coverage) among the sets in T.

The following theorem captures the above algorithm.

Theorem 38. *There exists a randomized one-pass, $\tilde{O}(k^2)$-space, algorithm that with high probability finds a $O(\log\min(k,r))$ approximation of* Max-k-UniqueCover.

Proof. It has been shown in previous work [16, 120] that T is a $2+\epsilon$ approximation of Max-k-Cover. Demaine et al. [49] proved that Q is an $O(\log\min(k,r))$ approximation of Max-k-UniqueCover. In fact, they presented a polynomial time algorithm to find Q from T such that the number of uniquely covered elements is at least

$$\Omega(1/\log k)\cdot|\cup_{A\in T}A| \geq \Omega(1/\log k)\cdot 1/2\cdot\mathrm{OPT}^* \geq \Omega(1/\log k)\cdot\mathrm{OPT}\ .$$

We note that $\mathrm{OPT}^* \leq k\,\mathrm{OPT}$. Otherwise, one can find a set that covers more than OPT elements which is a contradiction.

The above algorithm needs to keep track of the elements being covered by T at all points during the stream. This requires $\tilde{O}(\mathrm{OPT}^*) = \tilde{O}(k\,\mathrm{OPT})$ space. Furthermore, storing the sets in T needs $\tilde{O}(k\,\mathrm{OPT})$ space. Finally, guessing z^* entails a $O(\epsilon^{-1}\log \mathrm{OPT})$ factor. Thus, the algorithm uses $\tilde{O}(\epsilon^{-1}k\,\mathrm{OPT})$ space which could be translated into another algorithm that uses $\tilde{O}(\epsilon^{-3}k^2)$ space after using the subsampling framework. For the purpose of the proving the claimed approximation factor we can set ϵ to a small constant. □

Application to Parameterized Set Cover.

We parameterize the set cover problem as follows. Given a set system, either A) output a set cover of size αk if $\mathrm{OPT} \leq k$ where α the approximation factor or B) correctly declare that a set cover of size k does not exist.

Theorem 39. *There exists a randomized, $O(1/\delta)$-pass, $\tilde{O}(rk^2n^{1/\delta}+n)$-space, algorithm that with high probability finds a $O(1/\delta)$ approximation of the parameterized set cover problem.*

Proof. In each pass, we run the algorithm in Theorem 36 with parameters k and $\epsilon = 1/n^{\delta/3}$ on the remaining uncovered elements. The space use is $\tilde{O}(rk^2n^{1/\delta}+n)$. Here, we need additional $\tilde{O}(n)$ space to keep track of the remaining uncovered elements.

Note that if $\mathrm{OPT} \leq k$, after each pass, the number of uncovered elements is reduced by a factor $1/n^{\delta/3}$. This is because if n' is the number of uncovered elements at the beginning of a pass, then after that pass, we cover all but at most $n'/n^{\delta/3}$ of those elements. After i passes, the number of remaining uncovered elements is $O(n^{1-i\delta/3})$; we therefore use at most $O(1/\delta)$ passes until we are done. At the end, we have a set cover of size $O(k/\delta)$.

If after $\omega(1/\delta)$ passes, there are still remaining uncovered elements, we declare that such a solution does not exist. □

Our algorithm improves upon the algorithm by Har-Peled et al. [78] that uses $\tilde{O}(mn^{1/\delta}+n)$ space for when $rk^2 \ll m$ and also yields an $O(1/\delta)$ approximation.

Extension to Insert/Delete Streams

The result can be extended to the case where sets are inserted and deleted using the same approach as that used for unique coverage.

3.5 Lower Bounds

3.5.1 Lower Bounds for Exact Solutions

As observed earlier, any exact algorithm for either the Max-k-Cover or Max-k-UniqueCover problem on an input where all sets have size d will return a matching of size k if one exists. However, by a lowerbound due to Chitnis et al. [38] we know that determining if there exists a matching of size k in a single pass requires $\Omega(k^d)$ space. This immediately implies the following theorem.

Theorem 40. *Any single-pass algorithm that solves* Max-k-Cover *or* Max-k-UniqueCover *exactly with probability at least $9/10$ requires $\Omega(k^d)$ space.*

3.5.2 Lower bound for a $e^{1-1/k}$ approximation

The strategy is similar to previous work on Max-k-Cover [119, 120]. However, we need to argue that the relevant probabilistic construction works for all collections of fewer than k sets since the unique coverage function is not monotone. This extra argument will also allow us to show that the lower bound also applies to bi-criteria approximation in which we are allowed to pick more than k sets (this is not the case for Max-k-Cover).

We make a reduction from the communication problem k-player set disjointness, denoted by $\mathsf{DISJ}(m, k)$. In this problem, there are k players where the ith player has a set $S_i \subseteq [m]$. It is promised that exactly one of the following two cases happens a) NO instance: All the sets are pairwise disjoint and b) YES instance: There is a unique element $v \in [m]$ such that $v \in S_i$ for all $i \in [k]$ and all other elements belong to at most one set. The (randomized) communication complexity, for some large enough constant success probability, of the above problem in p-round, one-way model is $\Omega(m/(pk))$ even if the players may use public randomness [33]. We can assume that $|S_1 \cup S_2 \cup \ldots \cup S_k| \geq m/4$ via a padding argument.

Theorem 41. *Any constant-pass randomized algorithm with an approximation better than $e^{1-1/k}$ for* Max-k-UniqueCover *requires $\Omega(m/k^2)$ space.*

Proof. Consider a sufficiently large n where k divides n. For each $i \in [m]$, let $\mathcal{P}_i$ be a random partition of $[n]$ into k sets $V_1^i, \ldots, V_k^i$ such that an element in the universe $U = [n]$ belongs to exactly one of these sets uniformly at random. In particular, for all $i \in [m]$ and $v \in U$,

$$\mathbb{P}\left[v \in V_j^i \wedge (\forall j' \neq j, v \notin V_{j'}^i)\right] = 1/k\,.$$

The partitions are chosen independently using public randomness before receiving the input. For each player j, if $i \in S_j$, then they put V_j^i in the stream. Note that the stream consists of $\Theta(m)$ sets.

If the input is a NO instance, then for each $i \in [m]$, there is at most one set V_j^i in the stream. Therefore, for each element $v \in [n]$ and any collection of $\ell \leq k$ sets $V_{j_1}^{i_1}, \ldots, V_{j_\ell}^{i_\ell}$ in the stream,

$$\begin{aligned}\mathbb{P}\left[v \text{ is uniquely covered by } V_{j_1}^{i_1}, \ldots, V_{j_\ell}^{i_\ell}\right] &= \ell/k \cdot (1-1/k)^{\ell-1} \\ &\leq \ell/k \cdot e^{-(\ell-1)/k}\,.\end{aligned}$$

Therefore, in expectation, $\mu_\ell := \mathbb{E}\left[h(\{V_{j_1}^{i_1}, \ldots, V_{j_\ell}^{i_\ell}\})\right] \leq \ell/k \cdot e^{-(\ell-1)/k} n$ where $h()$ is the number of elements that are uniquely covered. By an application of Hoeffding's inequality,

$$\begin{aligned}&\mathbb{P}\left[h(\{V_{j_1}^{i_1} \cup \ldots \cup V_{j_\ell}^{i_\ell}\}) > \mu_\ell + \epsilon e^{-(k-1)/k} \cdot n\right] \\ \leq &\exp\left(-2\epsilon^2 e^{-2(\ell-1)/k} n\right) \\ \leq &\exp\left(-\Omega(\epsilon^2 n)\right) \leq \frac{1}{m^{10k}}\,.\end{aligned}$$

The last inequality follows by letting $n = \Omega(\epsilon^{-2} k \log m)$. The following claim shows that for large k, in expectation, picking k sets is optimal in terms of unique coverage.

Lemma 42. *The function $g(\ell) = \ell/k \cdot e^{-(\ell-1)/k} n$ is increasing in the interval $(-\infty, k]$ and decreasing in the interval $[k, +\infty)$.*

Proof. We take the partial derivative of g with respect to ℓ

$$\frac{\partial g}{\partial \ell} = \frac{e^{(1-\ell)/k}(k-\ell)}{k^2} \cdot n$$

and observe that it is non-negative if and only if $\ell \leq k$. □

By appealing to the union bound over all $\binom{m}{1} + \ldots + \binom{m}{k-1} + \binom{m}{k} \leq O(m^{k+1})$ possible collections $\ell \leq k$ sets, we deduce that with high probability, for all collections of $\ell \leq k$ sets $S_1, \ldots, S_\ell$,

$$\begin{aligned} h(\{S_1, \ldots, S_\ell\}) &\leq \mu_\ell + \epsilon e^{-(k-1)/k} \cdot n \\ &\leq \ell/k \cdot e^{-(\ell-1)/k} n + \epsilon e^{-(k-1)/k} \cdot n \\ &\leq (1+\epsilon) e^{-1+1/k} n \ . \end{aligned}$$

If the input is a YES instance, then clearly, the maximum k-unique coverage is n. This is because there exists i such that $i \in S_1 \cap \ldots \cap S_k$ and therefore $V_1^i, \ldots, V_k^i$ are in the stream and these sets uniquely cover all elements.

Therefore, any constant pass algorithm that finds a $(1+2\epsilon)e^{1-1/k}$ approximation of Max-k-UniqueCover for some large enough constant success probability implies a protocol to solve DISJ(m,k). Thus, $\Omega(m/k^2)$ space is required. □

Remark. Since $g(\ell)$ is decreasing in the interval $[k, m]$, the lower bound also holds for bi-criteria approximation where the algorithm is allows to pick more than k sets.

3.5.3 Lower bound for $1+\epsilon$ approximation

Assadi [14] presents a $O\ (m/\epsilon^2)$ lower bound for the space required to compute a $1+\epsilon$ approximation for Max-k-Cover when $k = 2$, even when the stream is in a random order and is allowed constant passes. This is accomplished via a reduction to multiple instances of the Gap-Hamming Distance problem on a hard input distribution, where an input with high maximum coverage corresponds to a YES answer for some Gap-Hamming Distance instance, and a low maximum coverage corresponds to a NO answer for all GHD instances. This hard distribution has the additional property that high maximum coverage inputs also have high maximum unique coverage, and low maximum coverage inputs have low maximum unique coverage. Therefore, the following corollary holds:

Corollary 43. *Any constant-pass randomized algorithm with an approximation factor* $1+\epsilon$ *for* Max-k-UniqueCover *requires* $\Omega(m/\epsilon^2)$ *space.*

CHAPTER 4

TEMPORAL GRAPH STREAMS

Temporal graphs incorporate the notion of time into the structure of traditional graphs to model time-dependent phenomena in applications such as information sharing, disease spreading, Internet routing, and more. In a temporal graph, edges are augmented with timestamps: labels which indicate at which moments in time the edge can be said to exist. Temporal graphs are a recent topic of study and no one has yet considered the problem of designing algorithms to compute properties of temporal graphs at scale. We introduce the notion of a temporal graph stream, where a temporal graph is defined via a sequence of temporal edge updates, and ask whether various properties of a temporal graph can be computed given sequential access to the stream and space sublinear in the size of the stream.

The significance and interpretation of edge timestamps may appear to vary with the algorithmic problem being considered, as well as the intended application. For instance, in this chapter we concern ourselves with reachability, where timestamps indicate moments at which the edge is traversable. In contrast, Mikhail et al. [124] consider the problem of finding a maximum matching where all edges in the matching have distinct timestamps. However, the idea that timestamps represent times at which an edge is traversible is central to existing work. For instance, the above temporal constraint on maximum matching is motivated as a technique for algorithms computing temporal versions of traveling salesman problems, which require a matching of temporally distinct edges to construct an approximately optimal traveling salesman tour. From this it is clear that connectivity in temporal graphs is a fundamental problem and a natural first topic of study.

We begin our study of temporal reachability in the data stream setting. In the non-temporal graph setting, a graph is connected iff there is a path between any two nodes in the graph. In temporal graphs, the fact that edges exist at some times and not others complicates our notion of a path. We consider *journeys*, paths on a temporal graph whose edges have strictly increasing timestamps. Given access to a stream defining T and $O(n \operatorname{polylog}(n))$ space, can we determine whether T has a journey between arbitrary nodes $(s, t) \in V$?

We begin investigating this question by considering various simpler versions of it. In particular we consider *forward* reachability where the task is to find all nodes reachable from some source node, and *backward* reachability where the task is to find all nodes from which some destination node is reachable. In Section 4.2, we prove lower space bounds on the problem of detecting forward journeys. In Section 4.3, we provide sketching algorithms for tracing randomly sampled backward journeys and estimating the number of backward journeys, and prove lower bounds for some other backwards reachability variants. In Section 4.4 we consider a generalization of the above problems where we trace journeys that depart before a certain timestep, and prove a strong lower bound on this generalized problem.

4.1 Preliminaries

We borrow some notation and definitions from [123].

Definition 44. *A* temporal graph T *is an ordered pair of disjoint sets* (V, A) *such that* $A \subset \binom{V}{2} \times \mathbb{N}$. *We refer to* A *as the set of* time-edges *of* T. *For* $e = (uv, t) \in A$, *we refer to* t *as the* timestamp *of* e. *Let* τ *denote the number of distinct timestamps in* A. *A* directed temporal graph D *is an ordered pair of disjoint sets* (V, A') *such that* $A' \subset V \times V \times \mathbb{N}$. *For* $e = (u, v, t) \in A'$, *we refer to to* u *as the* tail *of* e *and* v *as the* head *of* e.

Throughout this chapter we refer to classical graphs whose edges do not vary with time as *static graphs*.

As in static graphs, the notion of a *path* and resulting ideas of reachability and connectivity are fundamental to the study of temporal graphs. We begin with an investigation of these concepts.

Definition 45. *A* temporal *(or* time-respecting*) walk* W *on* T *is an alternating sequence of nodes and times* $\{u_1, t_1, u_2, t_2, \ldots, t_{k-1}, u_k\}$ *where* $(u_i u_{i+1}, t_i) \in A \; \forall i \in [k-1]$ *and* $t_i < t_{i+1} \; \forall i \in [k-2]$. *We call* t_1 *the* departure time *of* W, t_{k-1} *its* arrival time*, and* $t_{k-1} - t_1 + 1$ *its* duration. *A* journey *(or* temporal/time-respecting path*)* J *is a temporal walk with pairwise distinct nodes. We say that a node* v *is* reachable *from node* u *iff there is a journey from* u *to* v.

Note that T can have a journey from u to v but not have a journey from v to u. We define the reachability matrix $R \in \{0, 1\}^{n \times n}$ of T such that $R_{i,j} = 1$ if there exists a journey from i to j and 0 otherwise.[1]

In the streaming temporal graph setting, we assume that T is revealed via a stream of time-edge updates. A *time-respecting* stream is one whose time-edges arrive in increasing timestamp order.

4.2 Forward Reachability Problems

We consider some basic reachability problems in the temporal streaming setting. Typically, this means computing some portion of the reachability matrix, when the portion to be computed may be revealed before or after the stream. A natural temporal reachability problem to consider is *forward* reachability, where we must determine the set of nodes reachable from node s in T. If time-edges in the temporal graph represent infection-spreading contact events between people in a disease model, we can think of this forward reachability problem as tracking the spread of infection starting from person s. This forward reachability problem is equivalent to computing $R_{s,*}$, the sth row of the reachability matrix.

Theorem 46. *Computing* $R_{s,*}$ *in a time-respecting stream requires* $\tilde{\Theta}(n)$ *space if* s *is known before the stream.*

[1] We adopt the convention that there is always a journey from a node to itself, so $R_{i,i} = 1 \; \forall i$.

Proof. A very simple algorithm suffices. Initialize matrix $M^s \in \{0,1\}^{n \times \tau}$ such that $M^s_{s,0} = 1$ and 0 everywhere else. When the algorithm finishes, $M^s_{*,i}$ will represent the state of $R_{s,*}$ during timestep i. Process the stream of edges as follows: When the time-edge (uv, i) arrives in stream, if it is the first time-edge with timestamp i, set $M^s_{*,i} = M^s_{*,i-1}$ and then delete $M^s_{*,i-2}$. Then set $M^s_{u,i} = \max\{M^s_{u,i}, M^s_{v,i-1}\}$ and $M^s_{v,i} = \max\{M^s_{v,i}, M^s_{u,i-1}\}$. Let t' be the timestamp of the last time-edge in the stream. At the end of the stream, return $M^s_{*,t'}$.

Proof of correctness follows by a simple induction argument on timesteps. Since we delete columns as we go, we only ever represent two columns at any time in memory and therefore use $O(n)$ space.

We now prove the lower bound via a reduction from the communication problem of indexing. See the proof of Theorem 2 in Chapter 2 for details on the indexing problem. Suppose that we have a streaming algorithm that computes the problem. Alice constructs an input for the algorithm as follows: First, she creates a graph on nodes $\{g\} \cup \mathcal{V} \cup \{c\}$ with $|\mathcal{V}| = n$, and indicates to the algorithm to compute $R_{g,*}$. Then for each $j \in [n]$, Alice adds $(gv_j, 1)$ to the stream iff $a_j = 1$ where a is her binary string. She then sends the contents of her memory to Bob who has index b. Bob adds time-edge $(v_b c, 2)$ to the stream and has it return $R_{g,c}$. This solves the index problem since $R_{g,c} = 1$ iff time-edge $(gv_b, 1)$ appeared in the stream which occurred iff $a_b = 1$. Since the indexing problem requires space $\Omega(n)$, this streaming algorithm must also use space $\Omega(n)$. □

We might next ask whether it is possible to compute $R_{s,*}$ in small space even when s is not revealed until after the stream. This is essentially equivalent to computing all of R, even if we are allowed to fail to find $R_{s,*}$ with constant probability (since we can simply repeat such a procedure $\log(n)$ times in parallel to acheive a failure probability polynomially small in n).

Conjecture 47. *Computing $R_{s,*}$ in a time-respecting stream with probability $3/4$ requires $\Omega(n^2)$ space if s is not known until after the stream.*

We now prove $\Omega(n^2)$ lower bounds for some generalizations of this problem. It turns out that for these generalizations, we can prove an $\Omega(n^2)$ lower bound for an even simpler task: computing $R_{s,z}$ for $s, z \in V$ even if both s and z known before the stream.

Theorem 48. *Computing $R_{s,z}$ of a directed temporal graph D in a time-respecting stream with success probability $9/10$ requires $\Omega(n^2)$ space even if s and z are known before the stream.*

Proof. We prove via a reduction from the index problem. Suppose there exists a streaming algorithm that computes the problem with probability at least $9/10$. Alice, who has binary string $a \in \{0,1\}^{n^2}$, constructs an input for the streaming algorithm as follows: First, she creates a graph on nodes $\{g\} \cup \mathcal{U} \cup \mathcal{V} \cup \{c\}$ with $|\mathcal{U}| = |\mathcal{V}| = n$, and indicates to the algorithm to compute $R_{g,c}$. Alice maps each index of a to a unique node pair in $\mathcal{U} \times \mathcal{V}$ via a one-to-one function $f : [n^2] \to [n] \times [n]$. For convenience, denote the nodes in this unique pair as $f_u(i)$ and $f_v(i)$ for each input i. For each $i \in [n^2]$, Alice adds directed time-edge $(g, u_i, 1)$ to the stream. Then for each $i \in [n]$, Alice adds directed time-edge $(f_u(i), f_v(i), 2)$ to the stream iff $a_i = 1$. She then sends the contents of her memory to Bob who has index

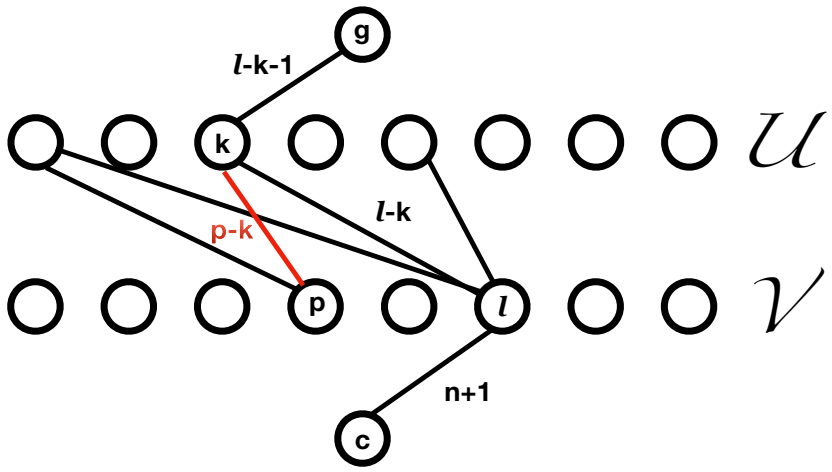


Figure 4.1: The construction used in the proof of Theorem 49. Note how any journey from u_k and v_l must have an edge with timestamp less than $l-k$, with the exception of edge $e = (u_k v_l, l-k)$. Therefore there is a journey from g to c iff e is added to the stream.

b. Bob adds time-edge $(f_v(b), c, 3)$ to the stream and asks the algorithm to return the value $R_{g,c}$. Note that there is at most 1 journey from g to c, which is $\{g, 1, f_u(b), 2, f_v(b), 3, c\}$ and this journey exists only when directed time-edge $(f_u(b), f_v(b), 2)$ has been added to the stream which occurs only when $a_b = 1$. Since solving the indexing problem with probability $9/10$ requires space $\Omega(n^2)$, our streaming algorithm must also require space $\Omega(n^2)$. □

The above proof relies on the fact that there are no journeys from nodes in $\mathcal{V}$ to nodes in $\mathcal{U}$. For the next problem we consider, this is not true and so we need to carefully define the index-to-node-pair function f to account for this.

Theorem 49. *Computing $R_{s,z}$ in a non-time-respecting stream with success probability $9/10$ requires $\Omega(n^2)$ space even if s and z are known before the stream.*

Proof. Once again we prove this via a reduction to the index problem, using nearly the same proof as that of Theorem 48. The idea is to construct a temporal graph such that there is a journey from g to c iff Alice inserts the time-edge that corresponds to Bob's index b.

Suppose there exists a streaming algorithm that computes the problem with probability at least $9/10$. Alice, who has binary string $a \in \{0,1\}^{n^2}$, constructs an input for the streaming algorithm as follows: First, she creates a graph on nodes $\{g\} \cup \mathcal{U} \cup \mathcal{V} \cup \{c\}$ with $|\mathcal{U}| = |\mathcal{V}| = 2n$, and indicates to the algorithm to compute $R_{g,c}$. Order the nodes in $\mathcal{U}$ and $\mathcal{V}$ arbitrarily, and let u_j denote the jth node of $\mathcal{U}$ in this ordering and likewise let v_j denote the jth node of $\mathcal{V}$. Alice maps each index i of a to a unique node pair in $\mathcal{U} \times \mathcal{V}$ via the functions $f(i) = \lfloor i/n \rfloor$ and $h(i) = \lfloor i/n \rfloor + i \bmod n + 1$. Note that $f(i) < h(i)\ \forall i \in [n^2]$. Then for each index i, if $a_i = 1$ Alice computes adds time-edge $e_i = (u_{f(i)} v_{h(i)}, h(i) - f(i))$ to the stream. Then Alice sends the contents of her memory to Bob who has index b. Bob adds time-edges $(g u_k, l-k-1)$ and $(v_l c, n+1)$ where $f(b) = k$ and $h(b) = l$. There is a journey from g to c iff there is a journey from u_k to v_l with departure time $l-k$ or greater. The following lemma establishes that such a journey exists iff $a_b = 1$, completing the proof.

Claim 4. *For any $u_k \in \mathcal{U}$ and $v_l \in \mathcal{V}$, there is at most one journey from u_k to v_l with departure time $\geq l-k$. If such a journey exists, it is the single time-edge $(u_k v_l, l-k)$.*

Proof. Recall that all journeys among nodes in $\mathcal{U}$ and $\mathcal{V}$ must be composed of edges of strictly increasing timestamps. Certainly if time-edge $(u_k v_l, l-k)$ exists then it is a journey from u_k to v_l. We will prove the lemma by demonstrating that any other path between u_k and v_l must include an edge with timestamp less than $l-k$ and therefore cannot be part of a journey from g to v_l.

Consider a journey ψ from u_k to v_l with more than one time-edge. Since the subgraph on $\mathcal{U} \cup \mathcal{V}$ is bipartite, ψ must begin with time-edge $(u_k v_p, p-k)$ for some node $v_p \in \mathcal{V} \setminus v_l$. Consider the next two time-edges in ψ, $(v_p u_q, q-p)$ and $(u_q v_r, r-q)$ for some $u_q \in \mathcal{U}$ and $v_r \in \mathcal{V}$. Since timestamps strictly increase along the journey, $r-q > q-p$ and so $r = (r-q) + (q-p) + p > p$. Repeating this argument for all subsequent pairs of edges in the journey yields $l > p$. As a result, $(u_k v_p, p-k)$ occurs before $(u_k v_l, l-k)$ and ψ's departure time is less than $l-k$.This idea is illustrated in Figure 4.1. □

Due to Alice's construction, time-edge $(u_k v_l, l-k)$ is added to the stream iff $a_b = 1$. □

4.3 Backwards Reachability Problems

We now consider the *backward* reachability problem, where given a node z we must determine the set of nodes $\mathcal{U}$ such that there exists a journey from u to z for all $u \in \mathcal{U}$. In our example disease-spreading interpretation, we can think of backward reachability as finding the set of all potential patient zeros who could have infected person z. Backward reachability is equivalent to computing $R_{*,z}$, the zth column of the reachability matrix.

Similar to Conjecture 47, computing $R_{*,z}$ is essentially equivalent to computing all of R. We first focus our attention on more modest goals: using sketches to estimate $F_0(R_{*,z})$ or to sample a nonzero element of $R_{*,z}$ uniformly at random. Then we will prove some lower bounds for generalizations of the backwards problem.

Consider an initially empty temporal graph T which will be defined by a time-respecting temporal graph stream, and let $T(t)$ denote T after the stream has delivered all time-edges with timestamp t or less. Denote its reachability matrix $R^T(t)$. $R^T(0) = I_n$, the identity matrix of size n. Given $R^T(t-1)$ and the $A(t)$, the set of time-edges with timestamp exactly t, we can perform set of simple linear operations on the columns of $R^T(t-1)$ to compute $R^T(t)$. Let $R^T_{i,j}(t)$ denote the value of $R^T(t)$ at row i and column j, and let $R^T_{i,*}(t)$ and $R^T_{*,j}(t)$ denote the ith row and jth column of $R^T(t)$ respectively.

Lemma 50. *Let $B^t(t)$ denote the Let $A_u(t) = \{v : (uv, t) \in A(t)\} \cup \{u\}$. $R^T_{*,u}(t) = \bigvee_{v \in A_u(t)} R^T_{*,v}(t-1)$ where $\bigvee$ denotes the bitwise OR operation.*

Proof. Denote the set for which there exists a journey from u to v with arrival time $\leq t'$ as $\psi(v, t')$. From the definition of journey we have

$$\psi(u,t) = \psi(u, t-1) \cup \bigcup_{v \in A_u(t)} \psi(v, t-1)$$

$R^T_{*,\gamma}(t')$ is a binary encoding of set $\psi(\gamma, t')$ and we can perform the union operation via bitwise OR on these binary encodings. □

We can essentially perform OR operations on the columns by summing them together. Define $B^T(0) = I_n$ and $B^T_{*,u}(i) = \sum_{v \in A_u(i)} B^T_{*,v}(i-1)\ \forall i \in [\tau]$. If we treat all nonzero elements of B^T as equal to 1, then we can simply sum the appropriate columns together instead of performing bitwise OR (treating any nonzero entry as equivalent to 1). This means we can sketch the columns of R^T of the matrix using any linear sketch and update by summing different column sketches. Using existing sketches, we can immediately perform tasks in small space like estimating $F_0(R_{*,z})$ or to sample a nonzero element of $R_{*,z}$ uniformly at random.

However, this summing approach introduces a complication we must address. Elements of $R^T(t)$ may be as large as 2^t. For example, if the stream contains time-edges (u, v, i) for each $i \in [\tau]$, then $R^T_{u,v}(t) = 2^\tau$. F_0 and ℓ_0 sketches (which we use below) use space dependent on $\log(\sigma)$ where σ is the maximum value which elements in the sketched vector can take on, so in our example the presence of large values would incur an additional τ space factor. However, [115] outlines alternative versions of F_0 and ℓ_0 sketches which sketch each vector element mod p for some large random prime $p \ll n$, eliminating this extra τ space factor and leading to the following theorem:

Theorem 51. *There is a sketch-based time-respecting temporal graph stream algorithm that uses $O(n\epsilon^{-2}\log(1/\delta))$ space that $(1+\epsilon)$-approximates $F_0(R_{*,z})$ with probability $1-\delta$, and a sketch-based time-respecting temporal graph stream algorithm using $O(n\,\mathrm{polylog}(n)\log(1/\delta))$ space that samples a nonzero element of $R_{*,z}$ with probability $1-\delta$.*

4.4 Departure Reachability

So far we have been considering journeys which may depart and arrive at any time. We might however want to determine whether a journey with a departure time greater than some constant k exists between some nodes u and v. To accomodate this, we define the *departure-reachability matrix* $\mathcal{D}$ of a temporal graph T to be an $n \times n \times \tau$ matrix where $\mathcal{D}_{u,v,t} = 1$ iff there is a journey from u to v with departure time t or greater, and 0 otherwise.

Theorem 52. *Computing $\mathcal{D}_{s,z,t}$ in a time-respecting stream with success probability $9/10$ requires $\Omega(n^2)$ space if t and s are not known before the stream, even if z is known.*

Proof. The proof is nearly identical to that of Theorem 49. We assume there exists a streaming algorithm that finds $D_{s,z,t}$. Alice follows exactly the same procedure to construct the temporal graph stream, except that she asks the algorithm to compute $\mathcal{D}_{s',c,t'}$ for some not-yet-known value of t' and some not-yet-specified node s'. Bob, instead of adding edge $(gu_k, l-k)$ to the stream, indicates that $t' = l-k$ and $s' = u_k$. Then by Claim 4 there is a journey from u_k to c with departure time $\geq l-k$ iff time-edge $(u_k v_l, l-k)$ is added to the stream, which by Alice's construction occurs iff $a_b = 1$. □

Algorithms in the Field

CHAPTER 5

MESH

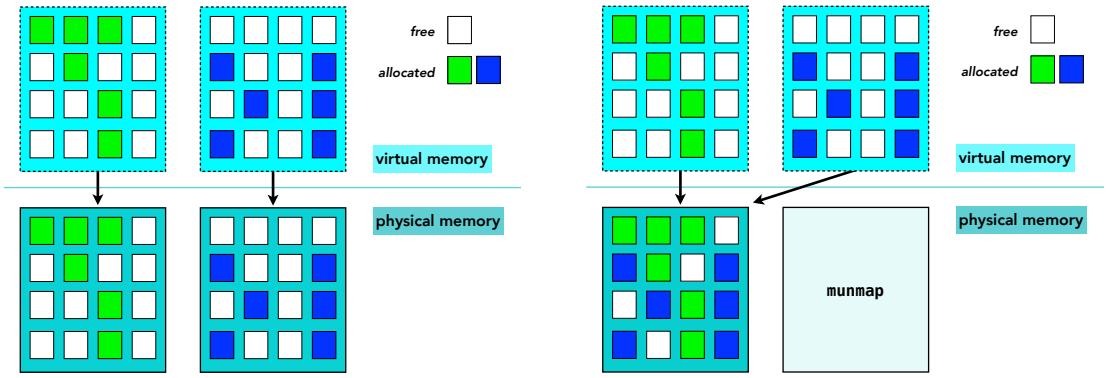
Programs written in C/C++ can suffer from serious memory fragmentation, leading to low utilization of memory, degraded performance, and application failure due to memory exhaustion. This chapter introduces MESH, a plug-in replacement for `malloc` that, for the first time, eliminates fragmentation in unmodified C/C++ applications. MESH combines novel randomized algorithms with widely-supported virtual memory operations to provably reduce fragmentation, breaking the classical Robson bounds with high probability. We focus here on the randomized algorithms which power MESH and proofs of their solution quality and runtime. MESH generally matches the runtime performance of state-of-the-art memory allocators while reducing memory consumption; in particular, it reduces the memory of consumption of Firefox by 16% and Redis by 39%.

5.1 Introduction

Despite nearly fifty years of conventional wisdom indicating that compaction is impossible in unmanaged languages, this chapter shows that it is not only possible but also practical. It introduces MESH, a memory allocator that effectively and efficiently performs compacting memory management to reduce memory usage in unmodified C and C++ applications.

Crucially and counterintuitively, MESH performs compaction without relocation; that is, without changing the addresses of objects. This property is vital for compatibility with arbitrary C/C++ applications. To achieve this, MESH builds on a mechanism which we call *meshing*, first introduced by Novark et al.'s Hound memory leak detector [131]. Hound employed meshing in an effort to avoid catastrophic memory consumption induced by its memory-inefficient allocation scheme, which can only reclaim memory when every object on a page is freed. Hound first searches for pages whose live objects do not overlap. It then copies the contents of one page onto the other, remaps one of the *virtual* pages to point to the single *physical* page now holding the contents of both pages, and finally relinquishes the other physical page to the OS. Figure 5.1 illustrates meshing in action.

MESH overcomes two key technical challenges of meshing that previously made it both inefficient and potentially entirely ineffective. First, Hound's search for pages to mesh involves a linear scan of pages on calls to `free`. While this search is more efficient than a naive $O(n^2)$ search of all possible pairs of pages, it remains prohibitively expensive for use in the context of a general-purpose allocator. Second, Hound offers no guarantees that *any* pages would ever be meshable. Consider an application that happens to allocate even one object in the same offset in every page. That layout would preclude meshing altogether, eliminating the possibility of saving any space.



(a) **Before:** these pages are candidates for "meshing" because their allocated objects do not overlap.

(b) **After:** both virtual pages now point to the first physical page; the second page is now freed.

Figure 5.1: **MESH in action.** MESH employs novel randomized algorithms that let it efficiently find and then "mesh" candidate pages within *spans* (contiguous 4K pages) whose contents do not overlap. In this example, it increases memory utilization across these pages from 37.5% to 75%, and returns one physical page to the OS (via `munmap`), reducing the overall memory footprint. MESH's randomized allocation algorithm ensures meshing's effectiveness with high probability.

MESH makes meshing both efficient and provably effective (with high probability) by combining it with two novel randomized algorithms. First, MESH uses a space-efficient randomized allocation strategy that effectively scatters objects within each virtual page, making the above scenario provably exceedingly unlikely. Second, MESH incorporates an efficient randomized algorithm that is guaranteed with high probability to quickly find candidate pages that are likely to mesh. These two algorithms work in concert to enable formal guarantees on MESH's effectiveness. Our analysis shows that MESH breaks the above-mentioned Robson worst case bounds for fragmentation with high probability [137], as memory reclaimed by meshing is available for use by any size class. This ability to redistribute memory from one size class to another enables Mesh to adapt to changes in an application's allocation behavior in a way other segregated-fit allocators cannot.

We implement MESH as a library for C/C++ applications running on Linux or Mac OS X. MESH interposes on memory management operations, making it possible to use it without code changes or recompilation by setting the appropriate environment variable to load the MESH library (e.g., `export LD_PRELOAD=libmesh.so` on Linux). Our evaluation demonstrates that our implementation of MESH is both fast and efficient in practice. It generally matches the performance of state-of-the-art allocators while guaranteeing the absence of catastrophic fragmentation with high probability. In addition, it occasionally yields substantial space savings: replacing the standard allocator with MESH automatically reduces memory consumption by 16% (Firefox) to 39% (Redis).

5.1.1 Contributions

This chapter describes the MESH system, focusing on its core meshing algorithm. It presents theoretical results that guarantee MESH's efficiency and effectiveness with high

probability (S5.4). Other components of the MESH system design and empirical evaluation of its performance are briefly summarized to contextualize the algorithm and analysis.

5.2 Overview

This section provides a high-level overview of how MESH works and gives some intuition as to how its algorithms and implementation ensure its efficiency and effectiveness, before diving into detailed description of MESH's algorithms (S5.3), implementation (S5.3.4), and its theoretical analysis (S5.4).

5.2.1 Remapping Virtual Pages

MESH enables compaction without relocating object addresses; it depends only on hardware-level virtual memory support, which is standard on most computing platforms like x86 and ARM64. MESH works by finding pairs of pages and merging them together *physically* but not *virtually*: this merging lets it relinquish physical pages to the OS.

Meshing is only possible when no objects on the pages occupy the same offsets. A key observation is that as fragmentation increases (that is, as there are more free objects), the likelihood of successfully finding pairs of pages that mesh also increases.

Figure 5.1 schematically illustrates the meshing process. MESH manages memory at the granularity of *spans*, which are runs of contiguous 4K pages (for purposes of illustration, the figure shows single-page spans). Each span only contains same-sized objects. The figure shows two spans of memory with low utilization (each is under 40% occupied) and whose allocations are at non-overlapping offsets.

Meshing consolidates allocations from each span onto one physical span. Each object in the resulting meshed span resides at the same offset as it did in its original span; that is, its virtual addresses are preserved, making meshing invisible to the application. Meshing then updates the virtual-to-physical mapping (the page tables) for the process so that both virtual spans point to the same physical span. The second physical span is returned to the OS. When average occupancy is low, meshing can consolidate many pages, offering the potential for considerable space savings.

5.2.2 Random Allocation

A key threat to meshing is that pages could contain objects at the same offset, preventing them from being meshed. In the worst case, all spans would have only one allocated object, each at the same offset, making them non-meshable. MESH employs randomized allocation to make this worst-case behavior exceedingly unlikely. It allocates objects uniformly at random across all available offsets in a span. As a result, the probability that all objects will occupy the same offset is $(1/b)^{n-1}$, where b is the number of objects in a span, and n is the number of spans.

In practice, the resulting probability of being unable to mesh many pages is vanishingly small. For example, when meshing 64 spans with one 16-byte object allocated on each (so that the number of objects b in a 4K span is 256), the likelihood of being unable to mesh any

of these spans is 10^{-152}. To put this into perspective, there are estimated to be roughly 10^{82} particles in the universe.

We use randomness to guide the design of MESH's algorithms (S5.3) and implementation (S5.3.4); this randomization lets us prove robust guarantees of its performance (S5.4), showing that MESH breaks the Robson bounds with high probability.

5.2.3 Finding Spans to Mesh

Given a set of spans, our goal is to mesh them in a way that frees as many physical pages as possible. We can think of this task as that of partitioning the spans into subsets such that the spans in each subset mesh. An optimal partition would minimize the number of such subsets.

Unfortunately, as we show, optimal meshing is not feasible (S5.4). Instead, the algorithms in Section 5.3 present practical methods for finding high-quality meshes under real-world time constraints. We show that solving a simplified version of the problem (S5.3) is sufficient to achieve reasonable meshes with high probability (S5.4).

5.3 Algorithms & System Design

MESH comprises three main algorithmic components: allocation (S5.3.1), deallocation (S5.3.2), and finding spans to mesh (S5.3.3). Unless otherwise noted and without loss of generality, all algorithms described here are per size class (within spans, all objects are same size).

5.3.1 Allocation

Allocation in MESH consists of two steps: (1) finding a span to allocate from, and (2) randomly allocating an object from that span. MESH always allocates from a thread-local shuffle vector – a randomized version of a freelist The shuffle vector contains offsets corresponding to the slots of a single span. We call that span the *attached* span for a given thread.

If the shuffle vector is empty, MESH relinquishes the current thread's attached span (if one exists) to the *global heap* (which holds all unattached spans), and asks it to select a new span. If there are no partially full spans, the global heap returns a new, empty span. Otherwise, it selects a partially full span for reuse. To maximize utilization, the global heap groups spans into bins organized by decreasing occupancy (e.g., 75-99% full in one bin, 50-74% in the next). The global heap scans for the first non-empty bin (by decreasing occupancy), and randomly selects a span from that bin.

Once a span has been selected, the allocator adds the offsets corresponding to the free slots in that span to the thread-local shuffle vector (in a random order). MESH pops the first entry off the shuffle vector and returns it.

```
SPLITMESHER(S, t)
1  n = length(S)
2  S_l, S_r = S[1 : n/2], S[n/2 + 1 : n]
3  for (i = 0, i < t, i++)
4       len = |S_l|
5       for (j = 0, j < len, j++)
6            if MESHABLE (S_l(j), S_r(j + i % len))
7                 S_l ← S_l \ S_l(j)
8                 S_r ← S_r \ S_r(j + i % len)
9                 MESH(S_l(j), S_r(j + i % len))
```

Figure 5.2: **Meshing random pairs of spans.** SPLITMESHER splits the randomly ordered span list S into halves, then probes pairs between halves for meshes. Each span is probed up to t times.

5.3.2 Deallocation

Deallocation behaves differently depending on whether the free is local (the address belongs to the current thread's attached span), remote (the object belongs to another thread's attached span), or if it belongs to the global heap.

For local frees, MESH adds the object's offset onto the span's shuffle vector in a random position and returns. For remote frees, MESH atomically resets the bit in the corresponding index in a bitmap associated with each span. Finally, for an object belonging to the global heap, MESH marks the object as free, updates the span's occupancy bin; this action may additionally trigger meshing.

5.3.3 Meshing

When meshing, MESH randomly chooses pairs of spans and attempts to mesh each pair. The meshing algorithm, which we call SPLITMESHER (Figure 5.2), is designed both for practical effectiveness and for its theoretical guarantees. The parameter t, which determines the maximum number of times each span is probed (line 3), enables space-time trade-offs. The parameter t can be increased to improve mesh quality and therefore reduce space, or decreased to improve runtime, at the cost of sacrificed meshing opportunities. We empirically found that $t = 64$ balances runtime and meshing effectiveness, and use this value in our implementation.

SPLITMESHER proceeds by iterating through S_l and checking whether it can mesh each span with another span chosen from S_r (line 6). If so, it removes these spans from their respective lists and meshes them (lines 7–9). SPLITMESHER repeats until it has checked $t * |S_l|$ pairs of spans.

5.3.4 Implementation

We implement MESH as a drop-in replacement memory allocator that implements meshing for single or multi-threaded applications written in C/C++. Its current implementation work for 64-bit Linux and Mac OS X binaries. MESH can be explicitly linked against

by passing `-lmesh` to the linker at compile time, or loaded dynamically by setting the `LD_PRELOAD` (Linux) or `DYLD_INSERT_LIBRARIES` (Mac OS X) environment variables to point to the MESH library. When loaded, MESH interposes on standard libc functions to replace all memory allocation functions.

MESH combines traditional allocation strategies with meshing to minimize heap usage. Like most modern memory allocators [21, 22, 59, 69, 130], MESH is a segregated-fit allocator. MESH employs fine-grained size classes to reduce internal fragmentation due to rounding up to the nearest size class. MESH uses the same size classes as those used by jemalloc for objects 1024 bytes and smaller [59], and power-of-two size classes for objects between 1024 and 16K. Allocations are fulfilled from the smallest size class they fit in (e.g., objects of size 33–48 bytes are served from the 48-byte size class); objects larger than 16K are individually fulfilled from the global arena. Small objects are allocated out of *spans* (S5.2), which are multiples of the page size and contain between 8 and 256 objects of a fixed size. Having at least eight objects per span helps amortize the cost of reserving memory from the global manager for the current thread's allocator.

Objects of 4KB and larger are always page-aligned and span at least one entire page. MESH does not consider these objects for meshing; instead, the pages are directly freed to the OS.

MESH's heap organization consists of four main components. *MiniHeaps* track occupancy and other metadata for spans. *Shuffle vectors* enable efficient, random allocation out of a MiniHeap. *Thread local heaps* satisfy small-object allocation requests without the need for locks or atomic operations in the common case. Finally, the *global heap* manages runtime state shared by all threads, large object allocation, and coordinates meshing operations. We omit detailing discussion of these components of MESH in this document.

5.4 Analysis

This section shows that the SPLITMESHER procedure described in S5.3.3 comes with strong formal guarantees on the *quality* of the meshing found along with bounds on its *runtime*. In situations where significant meshing opportunities exist (that is, when compaction is most desirable), SPLITMESHER finds with high probability an approximation arbitrarily close to $1/2$ of the best possible meshing in $O\left(n/q\right)$ time, where n is the number of spans and q is the global probability of two spans meshing.

To formally establish these bounds on quality and runtime, we show that meshing can be interpreted as a graph problem, analyze its complexity (S5.4.1), show that we can do nearly as well by solving an easier graph problem instead (S5.4.2), and prove that SPLITMESHER approximates this problem with high probability (S5.4.4).

5.4.1 Formal Problem Definitions

Since MESH segregates objects based on size, we can limit our analysis to compaction within a single size class without loss of generality. For our analysis, we represent spans as binary strings of length b, the maximum number of objects that the span can store. Each bit represents the allocation state of a single object. We represent each span π with string s such that $s\left(i\right) = 1$ if π has an object at offset i, and 0 otherwise.

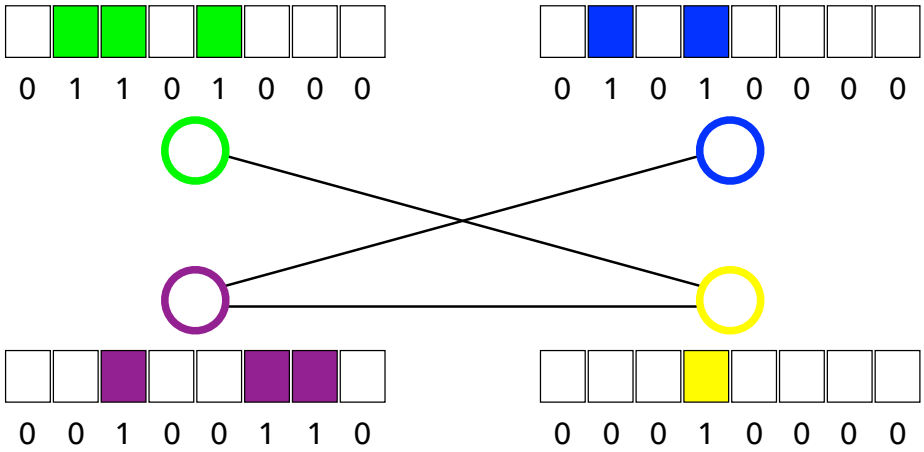


Figure 5.3: **An example meshing graph.** Nodes correspond to the spans represented by the strings `01101000`, `01010000`, `00100110`, and `00010000`. Edges connect meshable strings (corresponding to non-overlapping spans).

Definition 53. *We say two strings* s_1, s_2 mesh iff $\sum_i s_1(i) \cdot s_2(i) = 0$. More generally, a set of binary strings are said to mesh if every pair of strings in this set mesh.

When we mesh k spans together, the objects scattered across those k spans are moved to a single span while retaining their offset from the start of the span. The remaining $k-1$ spans are no longer needed and are released to the operating system. We say that we "release" $k-1$ *strings* when we mesh k strings together. Since our goal is to empty as many physical spans as possible, we can characterize our theoretical problem as follows:

Problem 1. *Given a multi-set of* n *binary strings of length* b*, find a meshing that releases the maximum number of strings.*

A Formulation via Graphs: We observe that an instance of the meshing problem, a string multi-set S, can naturally be expressed via a graph $G(S)$ where there is a node for every string in S and an edge between two nodes iff the relevant strings can be meshed. Figure 5.3 illustrates this representation via an example.

If a set of strings are meshable, then there is an edge between every pair of the corresponding nodes: the set of corresponding nodes is a *clique*. We can therefore decompose the graph into k disjoint cliques iff we can free $n-k$ strings in the meshing problem. Unfortunately, the problem of decomposing a graph into the minimum number of disjoint cliques (MINCLIQUECOVER) is in general NP-hard. Worse, it cannot even be approximated up to a factor $m^{1-\epsilon}$ unless $P = NP$ [151].

While the meshing problem is reducible to MINCLIQUECOVER, we have not shown that the meshing problem is NP-Hard. The meshing problem is indeed NP-hard for strings of arbitrary length, but in practice string length is proportional to span size, which is constant.

Theorem 54. *The meshing problem for* S*, a multi-set of strings of constant length, is in* P*.*

Proof. We assume without loss of generality that S does not contain the all-zero string s_0; if it does, since s_0 can be meshed with any other string and so can always be released, we can solve the meshing problem for $S \setminus s_0$ and then mesh each instance of s_0 arbitrarily.

Rather than reason about MINCLIQUECOVER on a meshing graph G, we consider the equivalent problem of coloring the complement graph $\bar{G}$ in which there is an edge between every pair of two nodes whose strings do not mesh. The nodes of $\bar{G}$ can be partitioned into at most $2^b - 1$ subsets $N_1 \ldots N_{2^b-1}$ such that all nodes in each N_i represent the same string s_i. The induced subgraph of N_i in $\bar{G}$ is a clique since all its nodes have a 1 in the same position and so cannot be pairwise meshed. Further, all nodes in N_i have the same set of neighbors.

Since N_i is a clique, at most one node in N_i may be colored with any color. Fix some coloring on $\bar{G}$. Swapping the colors of two nodes in N_i does not change the validity of the coloring since these nodes have the same neighbor set. We can therefore unambiguously represent a valid coloring of $\bar{G}$ merely by indicating in which cliques each color appears.

With 2^b cliques and a maximum of n colors, there are at most $(n+1)^c$ such colorings on the graph where $c = 2^{2^b}$. This follows because each color used can be associated with a subset of $\{1, \ldots, 2^b\}$ corresponding to which of the cliques have node with this color; we call this subset a *signature* and note there are c possible signatures. A coloring can be therefore be associated with a multi-set of possible signatures where each signature has multiplicity between 0 and n; there are $(n+1)^c$ such multi-sets. This is polynomial in n since b is constant and hence c is also constant. So we can simply check each coloring for validity (a coloring is valid iff no color appears in two cliques whose string representations mesh). The algorithm returns a valid coloring with the lowest number of colors from all valid colorings discovered. □

Note that the runtime of the above algorithm is at least exponential in the string length. While technically polynomial for constant string length, the running time of the above algorithm would obviously be prohibitive in practice and so we never employ it in MESH. Fortunately, as we show next, we can exploit the randomness in the strings to design a much faster algorithm.

5.4.2 Simplifying the Problem: From MINCLIQUECOVER to MATCHING

We leverage MESH's random allocation to simplify meshing; this random allocation implies a distribution over the graphs that exhibits useful structural properties. We first make the following important observation:

Observation 1. *Conditioned on the occupancies of the strings, edges in the meshing graph are not three-wise independent.*

To see that edges are not three-wise independent consider three random strings s_1, s_2, s_3 of length 4, each with exactly 2 ones. It is impossible for these strings to all mesh mutually since if we know that s_1 and s_2 mesh, and that s_2 and s_3 mesh, we know for certain that s_1 and s_3 cannot mesh. More generally, conditioning on s_1 and s_2 meshing and s_1 and s_3 meshing decreases the probability that s_1 and s_3 mesh. Below, we quantify this effect to argue that we can mesh near-optimally by solving the much easier MATCHING problem on the meshing graph (i.e., restricting our attention to finding cliques of size 2) instead of MINCLIQUECOVER. Another consequence of the above observation is that we will not be able to appeal to theoretical results on the standard model of random graphs, *ErdHos-Renyi*

graphs, in which each possible edge is present with some fixed probability and the edges are fully independent. Instead we will need new algorithms and proofs that only require independence of acyclic collections of edges.

5.4.2.1 Triangles and Larger Cliques are Uncommon.

Because of the dependencies across the edges present in a meshing graph, we can argue that *triangles* (and hence also larger cliques) are relatively infrequent in the graph and certainly less frequent than one would expect were all edges independent. For example, consider three strings $s_1, s_2, s_3 \in \{0,1\}^b$ with occupancies r_1, r_2, and r_3, respectively. The probability they mesh is

$$\binom{b-r_1}{r_2} / \binom{b}{r_2} \times \binom{b-r_1-r_2}{r_3} / \binom{b}{r_3}.$$

This value is significantly less than would have been the case if the events corresponding to pairs of strings being meshable were independent. For instance, if $b = 32, r_1 = r_2 = r_3 = 10$, this probability is so low that even if there were 1000 strings, the expected number of triangles would be less than 2. In contrast, had all meshes been independent, with the same parameters, there would have been 167 triangles.

The above analysis suggests that we can focus on finding only cliques of size 2, thereby solving MATCHING instead of MINCLIQUECOVER. The evaluation in Section 5.4.3 vindicates this approach, and we show a strong accuracy guarantee for MATCHING in Section 5.4.4.

5.4.3 Experimental Confirmation of Maximum Matching/Min Clique Cover Convergence

In Section 5.4.2, we argue that we can approximate the solution to MINCLIQUECOVER on meshing graphs with high probability by instead solving MAXIMUMMATCHING.

We experimentally verify this result by generating many random constant occupancy graphs and, for each graph, comparing the size of the maximum matching to the size of a greedy (non-optimal) solution for MINCLIQUECOVER. The results are summarized in Figure 5.4.

When we instead assume bits are 1 independently with probability p, we expect the graph to have many more triangles. For $p = r/b = 10/32, n = 1000$, the expected number of triangles is roughly 36,000. However, we can see experimentally that these graphs behave quite similarly in Figure 5.5.

While constant occupancy graphs are fairly regular, independent bit graphs may not be. Since strings have different occupancies, nodes which correspond to strings with relatively low occupancy will tend to have significantly higher degree than other nodes in the graph. Meanwhile, other nodes may have strings with high occupancy, and therefore only have a few edges (probably with low-occupancy nodes). So while there are many triangles, when the graph is sparse enough, even meshing cliques of size 3 and 4 will likely "abandon" adjacent high-occupancy nodes, one of which could have been matched with the high degree node to yield the same number of releases.

So in this case we still expect finding the maximum matching to be good enough.

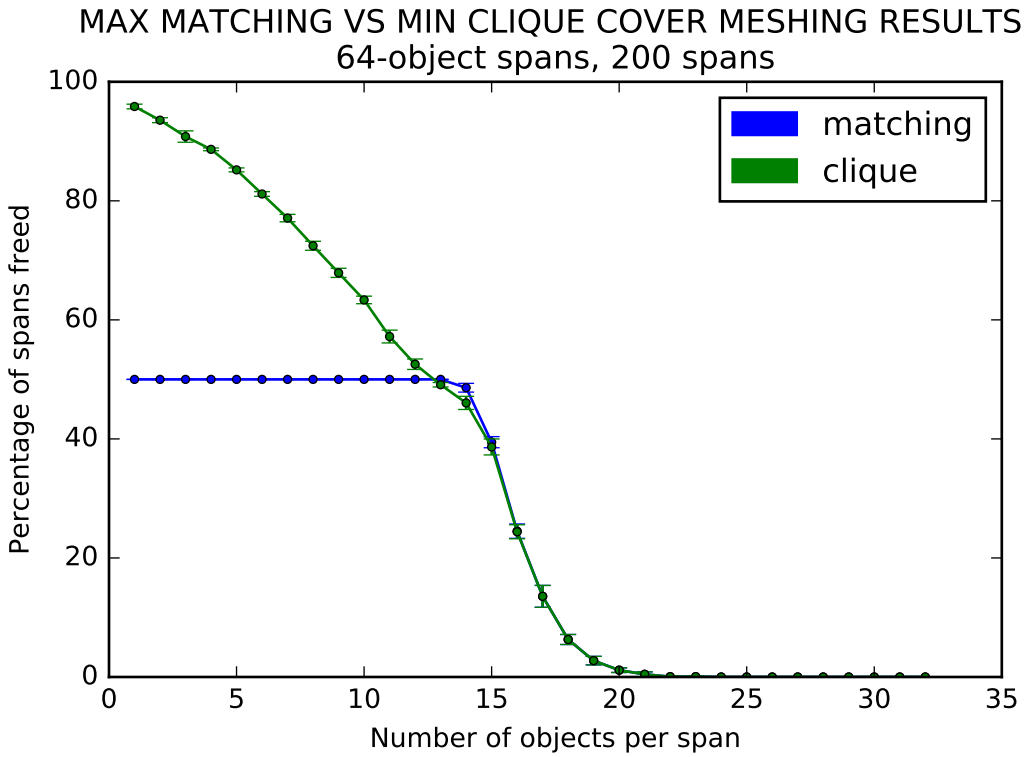


Figure 5.4: **Min Clique Cover and Max Matching solutions converge.** The average size of Min Clique Cover and Max Matching for randomly generated constant occupancy meshing graphs, plotted against span occupancy. Note that for sufficiently high-occupancy spans, Min Clique Cover and Max Matching are nearly equal.

5.4.4 Theoretical Guarantees

Since we need to perform meshing at runtime, it is essential that our algorithm for finding strings to mesh be as efficient as possible. It would be far too costly in both time and memory overhead to actually construct the meshing graph and run an existing matching algorithm on it. Instead, the SPLITMESHER algorithm (shown in Figure 5.2) performs meshing without the need for explicitly constructing the meshing graph.

For further efficiency, we need to constrain the value of the parameter t, which controls MESH's space-time tradeoff. If t were set as large as n, then SPLITMESHER could, in the worst case, exhaustively search all pairs of spans between the left and right sets: a total of $n^2/4$ probes. In practice, we want to choose a significantly smaller value for t so that MESH can always complete the meshing process quickly without the need to search all possible pairs of strings.

Lemma 55. *Let $t = k/q$ where $k > 1$ is some user defined parameter and q is the global probability of two spans meshing.* SPLITMESHER *finds a matching of size at least $n(1 - e^{-2k})/4$ between the left and right span sets with probability approaching 1 as $n \geq 2k/q$ grows.*

Proof. Let $S_l = \{v_1, v_2, \ldots v_{n/2}\}$ and $S_r = \{u_1, u_2, \ldots u_{n/2}\}$. Let $t = k/q$ where $k > 1$ is some arbitrary constant. For $u_i \in S_l$ and $i \leq j \leq j + t$, we say (u_i, v_j) is a *good match* if all the following properties hold: (1) there is an edge between u_i and v_j, (2) there are no

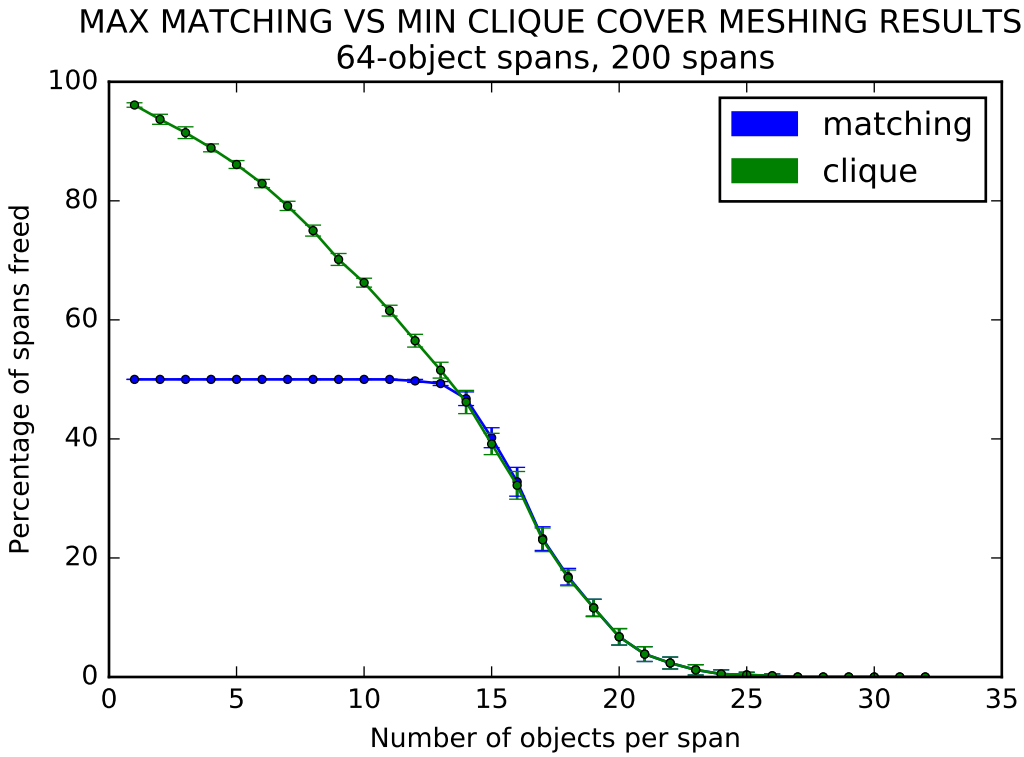


Figure 5.5: **Converge still holds for independent bits assumption.** The average size of Min Clique Cover and Max Matching for randomly generated constant occupancy meshing graphs, plotted against span occupancy. Note that for sufficiently high-occupancy spans, Min Clique Cover and Max Matching are nearly equal.

edges between u_i and $v_{j'}$ for $i \leq j' < j$, and (3) there are no edges between $u_{i'}$ and v_j for $i < i' \leq j$.

We observe that SPLITMESHER finds any good match, although it may also find additional matches. It therefore suffices to consider only the number of good matches. The probability (u_i, v_j) is a good match is $q(1-q)^{2(j-i)}$ by appealing to the fact that the collection of edges under consideration is acyclic. Hence, $\Pr(u_i \text{ has a good match})$ is

$$r := q \sum_{i=0}^{k/q-1} (1-q)^{2i} = q\frac{1-(1-q)^{2k/q}}{1-(1-q)^2} > \frac{1-e^{-2k}}{2} .$$

To analyze the number of good matches, define $X_i = 1$ iff u_i has a good match. Then, $\sum_i X_i$ is the number of good matches. By linearity of expectation, the expected number of good matches is $rn/2$. We decompose $\sum_i X_i$ into

$$Z_0 + Z_1 + \ldots + Z_{t-1} \quad \text{where} \quad Z_j = \sum_{i \equiv j \bmod t} X_i .$$

Since each Z_j is a sum of $n/(2t)$ independent variables, by the Chernoff bound, $\mathbb{P}\left[Z_j < (1-\epsilon)\, E[Z_j]\right] \leq \exp\left(-\epsilon^2 rn/(4t)\right)$. By the union bound,

$$\mathbb{P}\left[X < (1-\epsilon)\, rn/2\right] \leq t \cdot \exp\left(-\epsilon^2 rn/(4t)\right)$$

and this becomes arbitrarily small as n grows. □

In the worst case, the algorithm checks $nk/2q$ pairs. For our implementation of MESH, we use a static value of $t = 64$; this value enables the guarantees of Lemma 5.1 in cases where significant meshing is possible. As Section 5.5 shows, this value for t results in effective memory compaction with modest performance overhead.

5.4.5 New Lower Bound for Maximum Matching Size

In this section, we develop a bound for the size of the maximum matching in a graph that can easily be estimated in the context of meshing graphs. As meshing may be costly to perform, this lower bound is useful as it can be used to predict the magnitude of compaction achievable before committing to the process. In the case where little compaction is possible, it is often better not to try to mesh and instead conserve resources for other tasks. The quantity we introduce will always lower bound the size of the maximum matching and will typically be relatively close to the size of the maximum matching. For example, if we want to release 30% of our active spans through meshing, but the bound suggests a release of less than 5% is possible, we can infer that the maximum matching on the graph is small and meshing is currently not worth attempting.

Our approach is based on extending a result by McGregor and Vorotnikova [116]. Let $\mathrm{d}(u)$ be the degree of node u in a graph. They considered the quantity $\sum_{e\in E} 1/\max(\mathrm{d}(u), \mathrm{d}(v))$ and showed that it is at most a factor $3/2$ larger than the maximum matching in the graph and at most a factor 4 smaller in the case of planar graphs. These bounds were tight. For example, on a complete graph on three nodes, the quantity is $3/2$ while M is 1. Meshing graphs are very unlikely to be planar but are likely be almost regular, i.e., most degrees are roughly similar. We need to extend the above bound such that we can guarantee that it never exceeds the size of the maximum matching while also being a good estimate for the graphs that are likely to arise as meshing graphs.

One simple approach is to scale the above quantity by a factor of $2/3$, but this can result in a poor approximation for the size of the maximum matching for some graphs of interest. Instead, we take a more nuanced approach. Specifically, we prove the following theorem (proof omitted due to space constraints):

Theorem 56.

$$\mathcal{W} = \sum_{e\in E} \frac{1}{\max(\mathrm{d}(u), \mathrm{d}(v)) + I[\min(\mathrm{d}(u), \mathrm{d}(v)) > 1]} \leq M\ .$$

Proof. We begin by showing that the simpler quantity W is a lower bound of the maximum matching M.

$$W = \sum_{e\in E} \frac{1}{\max(\mathrm{d}\,(u)\,, \mathrm{d}\,(v)) + 1}\ .$$

Let U be an arbitrary set of t nodes in G where t is odd. Define

$$W(U) = \sum_{u,v\in U} \frac{1}{\max(\mathrm{d}\,(u)\,, \mathrm{d}\,(v)) + 1}\ .$$

As a corollary of Edmonds Matching Polytope Theorem, it can be shown that $W \leq M$ if $W(U) \leq (|U| - 1)/2$. We can argue this as follows:

$$
\begin{aligned}
W(U) &= \sum_{(u,v)\in U} \min\left(\frac{1}{\mathrm{d}(u)+1}, \frac{1}{\mathrm{d}(v)+1}\right) \\
&\leq \sum_{(u,v)\in U} \frac{1}{2}\left(\frac{1}{\mathrm{d}\,(u)+1} + \frac{1}{\mathrm{d}\,(v)+1}\right) \\
&\leq \sum_{(u,v)\in U} \frac{1}{2}\left(\frac{1}{\mathrm{d}_U\,(u)+1} + \frac{1}{\mathrm{d}_U\,(v)+1}\right) \\
&= \frac{1}{2}\sum_{u\in U} \frac{\mathrm{d}_U\,(u)}{\mathrm{d}_U\,(u)+1} \leq \frac{1}{2}\left(\frac{t-1}{t}\right) t = \frac{t-1}{2}
\end{aligned}
$$

where $\mathrm{d}_U(u)$ in the number of neighbors of u in the set U. The second line follows from the fact that the minimum of two quantities is bounded above by their average. The third line follows from the fact that the degree of any node in a subgraph is bounded above by its degree in the original graph. The fourth line follows from summing over nodes instead of edges, and then reasoning that in the worst case U is a clique and so $(u) = t - 1$ for all u u.

In some cases W is too conservative; it assigns little weight to edges which it could safely have assigned much more. For example, if e is isolated (meaning its endpoints have degree 1), $W(e) = 1/2$. However, it is always safe to assign weight 1 to e, since $M(G-e) = M(G) - 1$. If we modified our rule for W so that for any edge $e = (u, v)$ s.t. $\deg(u) = \deg(v) = 1$ we assigned weight $\min(1/\deg(u), 1/\deg(v))$ instead of $\min(1/\deg(u) + 1, 1/\deg(v) + 1)$, we would always assign weight 1 to isolated edges.

In fact, a more general rule is true. For any edge $e = (u, v)$, if either $\deg(u)$*or* $\deg(v) = 1$ then we may assign it weight $\min(1/\deg(u), 1/\deg(v))$.

Define $\mathcal{W}$ as follows:

$$
\mathcal{W} = \sum_{(u,v)\in E} \mathcal{W}(u, v)
$$

where

$$
\mathcal{W}(u, v) = \frac{1}{\max(\mathrm{d}\,(u)\,, \mathrm{d}\,(v)) + I[\min(\mathrm{d}(u), \mathrm{d}(v)) > 1]}
$$

We now show that $\mathcal{W} \leq M$. We have proven that $W(U) \leq (t-1)/2$ on any odd-size subgraph U, $|U| = t$. Define subsets U_1 and U_2 of U such that $U_1 \cup U_2 = U$. U_1 is the set of all nodes in U of degree 1 and all nodes in U adjacent to a node of degree 1, and U_2 is the set of all other nodes in U. Let $G(U_2)$ denote the subgraph of U induced by U_2, and let $|U_1| = x$ where x is even. Then $W(G(U_2)) = \mathcal{W}(G(U_2)) \leq (t - x - 1)/2$. So to complete our proof we must show that all remaining edges (call them E') have total weight $\leq x/2$.

Assume WLOG that there are no isolated edges in G (if there are, we can group them with $G(U_2)$ and retain the $(t - x - 1)/2$ bound).

$$\mathcal{W}(E') \leq \frac{1}{2} \sum_{u,v \in E'} \left(\frac{1}{\deg(u)} + \frac{1}{\deg(v)} \right)$$
$$= \sum_{k} \Big(\sum_{v \in U_1, \deg(v)=k} \frac{\delta}{k} + \frac{k-\delta}{2(k+1)} \Big)$$

where δ denotes the number of degree 1 nodes adjacent to v.
Let $f(\delta) = \delta/k + (k-\delta)/(2(k+1))$. We are interested in finding the maximum value $f(\delta)/(\delta+1)$ can take on; if it can never take a value greater than $1/2$ then $\mathcal{W}(E')$ cannot be greater than $x/2$.

$$\frac{\partial \frac{f(\delta)}{\delta+1}}{\partial \delta} = \frac{2-k}{2k(\delta+1)^2}$$

which is always negative for $k \geq 2$. So, $f(\delta)/(\delta+1)$ is maximized at $\delta = 0$, so $f(\delta)/(\delta+1) \leq k/(2(k+1)) < 1/2$.

There is one final detail we have not considered: x might be odd. In this case, $|U_2|$ is even and we can't appeal to Theorem 3 to say that $\mathcal{W}(G(U_2)) \leq (t-x-1)/2$. However, we can simply remove one node v' from U_2; the resulting odd-size subgraph has weight at most $(t-x-2)/2$. Since $\mathcal{W}$ is a valid fractional matching, the weight assigned to all edges adjacent to v' cannot exceed 1, so we can say that $\mathcal{W}(G(U_2)) \leq (t-x)/2$. Now we must show that $\mathcal{W}(E') \leq (x-1)/2$.

We have shown that the edge weight per node in U_1 cannot exceed $1/2$. $\mathcal{W}(E')$ is minimized when there is exactly 1 node of degree 1, with a degree k neighbor. In this case, the only edge in E' will be assigned weight $k/(2(k+1)) < 1/2$. $\mathcal{W}(E') \leq 1/2 + (x-2)/2 = (x-1)/2$ and the theorem is proven. □

A remark on estimating $\mathcal{W}$. If we wish to use this lower bound to decide whether to begin meshing by predicting the size of the maximum matching, we cannot compute it exactly because we do not know the degrees of the nodes in the graph. However, we do know the degree distribution of the graph and so it is possible to calculate the expected value of $\mathcal{W}$.

5.4.6 Summary of Analytical Results

We show the problem of meshing is reducible to a graph problem, MINCLIQUECOVER. While solving this problem is infeasible, we show that probabilistically, we can do nearly as well by finding the maximum MATCHING, a much easier graph problem. We analyze our meshing algorithm as an approximation to the maximum matching on a random meshing graph, and argue that it succeeds with high probability. Finally, we prove a new lower bound on the maximum matching for graphs based on degree distribution. As a corollary of these results, MESH breaks the Robson bounds with high probability.

5.5 Summary of Evaluation

For a number of memory-intensive applications, including aggressively space-optimized applications like Firefox, MESH can substantially reduce memory consumption (by 16% to 39%) while imposing a modest impact on runtime performance (e.g., around 1% for Firefox and SPECint 2006). We find that MESH's randomization can enable substantial space reduction in the face of a regular allocation pattern.

5.6 Related Work

Hound: Hound is a memory leak detector for C/C++ applications that introduced meshing (a.k.a. "virtual compaction"), a mechanism that MESH leverages [131]. Hound combines an age-segregated heap with data sampling to precisely identify leaks. Because Hound cannot reclaim memory until every object on a page is freed, it relies on a heuristic version of meshing to prevent catastrophic memory consumption. Hound is unsuitable as a replacement general-purpose allocator; it lacks both MESH's theoretical guarantees and space and runtime efficiency (Hound's repository is missing files and it does not build, precluding a direct empirical comparison here). The Hound paper reports a geometric mean slowdown of $\approx 30\%$ for SPECint2006 (compared to MESH's 0.7%), slowing one benchmark (`xalancbmk`) by almost $10\times$. Hound also generally *increases* memory consumption, while MESH often substantially decreases it.

Compaction for C/C++: Previous work has described a variety of manual and compiler-based approaches to support compaction for C++. Detlefs shows that if developers use annotations in the form of smart pointers, C++ code can also be managed with a relocating garbage collector [50]. Edelson introduced GC support through a combination of automatically generated smart pointer classes and compiler transformations that support relocating GC [53]. Google's Chrome uses an application-specific compacting GC for C++ objects called Oilpan that depends on the presence of a single event loop [2]. Developers must use a variety of smart pointer classes instead of raw pointers to enable GC and relocation. This effort took years. Unlike these approaches, MESH is fully general, works for unmodified C and C++ binaries, and does not require programmer or compiler support; its compaction approach is orthogonal to GC.

CouchDB and Redis implement *ad hoc* best-effort compaction, which they call "defragmentation". These work by iterating through program data structures like hash tables, copying each object's contents into freshly-allocated blocks (in the hope they will be contiguous), updating pointers, and then freeing the old objects [135, 139]. This application-specific approach is not only inefficient (because it may copy objects that are already densely packed) and brittle (because it relies on internal allocator behavior that may change in new releases), but it may also be ineffective, since the allocator cannot ensure that these objects are actually contiguous in memory. Unlike these approaches, MESH performs compaction efficiently and its effectiveness is guaranteed.

Compacting garbage collection in managed languages: Compacting garbage collection has long been a feature of languages like LISP and Java [65, 77]. Contemporary runtimes like the Hotspot JVM [125], the .NET VM [112], and the SpiderMonkey JavaScript VM [44] all implement compaction as part of their garbage collection algorithms. MESH brings the benefits of compaction to C/C++; in principle, it could also be used to automatically enable compaction for language implementations that rely on non-compacting collectors.

Bounds on Partial Compaction: Cohen and Petrank prove upper and lower bounds on defragmentation via partial compaction [41, 42]. In their setting, corresponding to managed environments, *every* object *may* be relocated to any free memory location; they ask what space savings can be achieved if the memory manager is only allowed to relocate a bounded number of objects. By contrast, MESH is designed for unmanaged languages where objects *cannot* be arbitrarily relocated.

PCM fault mitigation: Ipek *et al.* use a technique similar to meshing to address the degradation of phase-change memory (PCM) over the lifetime of a device [84]. The authors introduce dynamically replicated memory (DRM), which uses pairs of PCM pages with non-overlapping bit failures to act as a single page of (non-faulty) storage. When the memory controller reports a page with new bit failures, the OS attempts to pair it with a complementary page. A random graph analysis is used to justify this greedy algorithm.

DRM operates in a qualitatively different domain than MESH. In DRM, the OS occasionally attempts to pair newly faulty pages against a list of pages with static bit failures. This process is incremental and local. In MESH, the occupancy of spans in the heap is more dynamic and much less local. MESH solves a full, non-incremental version of the meshing problem each cycle. Additionally, in DRM, the random graph describes an error model rather than a design decision; additionally, the paper's analysis is flawed. The paper erroneously claims that the resulting graph is a simple random graph; in fact, its edges are not independent (as we show in S5.4.2). This invalidates the claimed performance guarantees, which depend on properties of simple random graphs. In contrast, we prove the efficacy of our original SPLITMESHER algorithm for MESH using a careful random graph analysis.

5.7 Conclusion

This chapter introduces MESH, a memory allocator that efficiently performs *compaction without relocation* to save memory for unmanaged languages. We show analytically that MESH provably avoids catastrophic memory fragmentation with high probability, and empirically show that MESH can substantially reduce memory fragmentation for memory-intensive applications written in C/C++ with low runtime overhead.

We have released MESH as an open source project; it can be used with arbitrary C and C++ Linux and Mac OS X binaries and can be downloaded at `http://libmesh.org`.

CHAPTER 6

PATHCACHE

Accurate prediction of network paths between arbitrary hosts on the Internet is of vital importance for network operators, cloud providers, and academic researchers. We present PathCache, a system that predicts network paths between arbitrary hosts on the Internet using historical knowledge of the data and control plane. In addition to feeding on freely available traceroutes and BGP routing tables PathCache uses graph algorithms to optimally explore network paths towards chosen BGP prefixes. PathCache's strategy for exploring network paths discovers 4X more autonomous systems (AS) hops than other well-known strategies used in practice today. Using a corpus of traceroutes, PathCache trains probabilistic models of routing towards all routed prefixes on the Internet and infers network paths and their likelihood. PathCache's AS-path predictions differ from the measured path by at most 1 hop, 75% of the time. A prototype of PathCache is live today to facilitate its inclusion in other applications and studies. We additionally demonstrate the utility of PathCache in improving real-world applications for circumventing Internet censorship and preserving anonymity online.

6.1 Contributions

This chapter describes the PathCache system, focusing on the algorithmic and data structural elements of its design and their analysis. We additionally include brief descriptions of PathCache's experimental evaluation and deployment for completeness.

PathCache Overview. Section 6.2 provides a summary of PathCache's architecture and explanation of several key design decisions.

Efficient global and per-destination topology discovery. A related system, Sibyl [98], allows efficient use of measurement budget for answering path queries between sources and destination IP addresses. Instead, PathCache focusses on a special case of this problem: efficiently answering *all* queries towards a destination prefix for a fixed measurement budget. PathCache's approach for doing measurement selection is within constant factor of optimal in the general case. However, when the routing towards a prefix is destination based, with no violations, we show that PathCache's measurement selection is optimal. Section 6.3 describes the topology discovery component of PathCache and establishes theoretical guarantees of its efficiency and optimality.

Per-destination probabilistic Markov models. Systems like iPlane and iPlane Nano consume traceroutes to build an atlas of network paths. By combining splices of paths from this atlas, the systems predict a previously un-measured path. Recently, the prediction

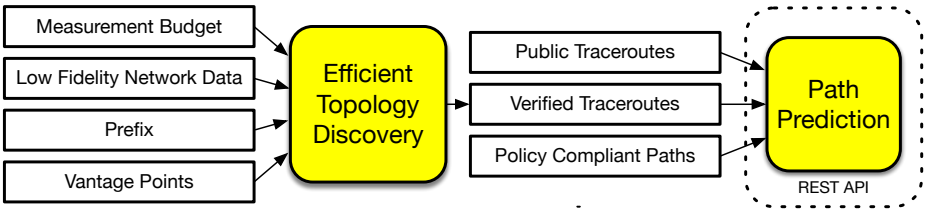


Figure 6.1: PathCache achieves two goals: efficient topology discovery and accurate path prediction.

accuracy of iPlane was found to be low (68% at the AS path level) by Sibyl [98]. Sibyl proposes to improve the low accuracy of splicing based path-prediction of network paths by using supervised learning for choosing between multiple possible paths.

PathCache differs from the existing approaches by using the key fact that routing on the Internet is largely destination based [12]. This means, for a given destination prefix, routes from a network are likely to traverse the same path, irrespective where they originated. Therefore, unlike previous path prediction systems, PathCache constructs a *destination-specific* probabilistic model for each destination prefix in place of a common atlas for all destination networks. Using observations of network paths over time, PathCache not only infers the connectivity between networks but also learns the likelihood of picking different next-hops from a given network. Section 6.4 describes this component of PathCache and outlines algorithmically efficient aspects of its design.

Evaluation and Deployment of PathCache. In Section 6.5, we briefly summarize the results of our experimental evaluation of PathCache. In Section 6.6, we describe our deployment of PathCache and describe its use in real-world applications. Finally, in Section 6.7, we explore a few possibilities for future work involving PathCache.

6.2 The PathCache System

Figure 6.1 shows the two major components of PathCache along with the input for each. These components are:

(1) *Efficient Topology Discovery.* PathCache makes efficient use of limited measurement resources to discover the network topology towards an IP prefix. PathCache's topology discovery algorithm takes as input a destination BGP prefix P, a set of vantage points V' capable of sending traceroutes to P, a measurement budget k, and information of P's network topology derived from BGP routing tables or older traceroutes (S6.3.1). This topology discovery algorithm outputs a budget-compliant set of vantage points $S \subset V', |S| \leq k$ from which to measure to reveal as much information about paths towards P as possible (S6.3.2). In S6.3, we describe this algorithm and its strong performance guarantees in detail.

(2) *Path Prediction.* PathCache combines public traceroutes from measurement platforms like RIPE Atlas [136] and those run by its own topology discovery module to learn Markov models of routing towards each BGP routed prefix (S6.4). PathCache infers network paths between source s and destination prefix P from the Markov model (S6.4). While PathCache aims to explore as much of the network topology as it can via active measurements, it will still lack a global view due to absence of vantage

points in different parts of the Internet. As a result, when a queried path (source s and destination P) cannot be inferred from P's empirically-based Markov model, PathCache will fall back to an algorithmic simulation of policy-compliant BGP paths [72].

6.2.1 Design Choices

We now give a brief overview of the design choices made while developing PathCache.

Granularity of predicted network paths. So far, we have not discussed what constitutes a network path predicted by PathCache. In previous research, systems for path prediction have attempted to predict paths at various granularity of intermediate hops. BGPSim [72] returns BGP policy compliant, AS-level paths, whereas iPlane [109], iNano [110], and Sibyl [98] predict PoP-level paths. The granularity of the predicted path can impact its utility for different applications. For instance, AS-level paths can be sufficiently informative for quantifying the threat of eavesdropping ISPs on anonymous communication [128, 143]. However, some ASes can be very large, spanning entire countries (Tier-1 networks like AT&T and Level3), making AS-level path prediction too coarse for diagnostic purposes. Inferring PoPs from router IP addresses is a research problem in its own right PathCache avoids introducing the complexity of PoP inference by predicting prefix-level paths. Prefix-level paths have sufficient information to predict AS-level paths by mapping prefixes back to ASes that announce them in BGP to obtain an AS-level path, if desired.

Markov model for path prediction. Since paths on the Internet are an outcome of several un-observable and uncertain phenomenon, it is natural to model their behavior using empirically derived probabilistic models. PathCache builds a Markov chain, one for each routed prefix P, using traceroutes that share a destination P. With the routed prefix as the *end state*, other routed prefixes on the Internet act as potential *start states* of the Markov chain, the problem of network path prediction is to find the most likely sequence of states from a given start state to the end state. PathCache additionally offers users the ability to predict paths based only on traceroutes performed during a specified window of time.

Granularity of destinations. While existing simulation approaches focus on paths towards destination ASes [72], we observed many violations of destination-based routing when considering destinations at the AS-level, since large ASes announce several prefixes, each getting routed to differently. This led us to our decision to construct Markov chains on a per-prefix basis. While BGP atoms [3] may have reduced the storage requirements of the per-prefix Markov chains, previous research [127] shows that over 70% of BGP atoms consist of only one IP prefix, limiting the potential reduction.

6.3 Efficient Topology Discovery

In this section, we discuss the problem of discovering the set of paths towards a destination prefix P. PathCache's measurement algorithm aims to maximize the amount of the Internet topology discovered when measuring paths towards P. It begins by using existing but imperfect data sources to construct a graph representation of the network topology. Exist-

ing data sources can be BGP paths, which are known to differ from data plane measurements or stale traceroutes which may not match the current network state (S6.3.1).

We frame the challenge of maximizing per-prefix topology discovery for a given measurement budget as an optimization problem, which we show is equivalent to a special case of the MAX-COVER$_k$ (S6.3.2) problem when routing is destination-based and present a greedy algorithm (S6.3.3) that optimally solves this topology discovery problem. In the case where networks violate destination-based routing, we can still guarantee a constant-factor approximation of the problem using a relaxation of the MAX-COVER$_k$ problem.

6.3.1 Existing Data Sources

We build an initial network topology of paths towards prefix P using the BGPSim [72] path simulator. BGPSim computes BGP policy compliant paths between any pair of ASes. So, for any prefix P, we find the policy compliant paths from all ASes on the Internet to the AS announcing prefix P. We note that this implies that the BGPSim derived topologies of all prefixes in the same AS are the same. The result of this computation is a tree of ASes rooted at the origin AS for P. If pre-existing traceroute data is used to augment the graph produced by BGPSim, the graph might have cycles when traceroutes include paths that existed at distinct time periods or when data- and control-plane paths do not agree. These cycles may also exist as a result of violations of destination-based routing [12]. We discuss cycles we observe and their implications on our results in Section 6.7.

6.3.2 Maximizing Topology Discovery

We define the destination-based DAG of a prefix P as $G = (V, E)$, where V consists of P and all ASes. Edge $e \in E$ represents an observed connection between two ASes. We consider the prefix as the root of this graph as opposed to the AS that announced it to account for per-prefix routing policies [12]. $V' \subset V$ is the set of ASes that have vantage points.

In practice, for a given prefix P, we do not know G. By executing traceroutes from a subset S of V', we obtain a *partial observation* of G denoted by $\hat{G} = (\hat{V}, \hat{E})$. $\hat{V}$ is composed of AS hops observed in traceroutes from ASes in V' towards P. Edges in $\hat{E}$ consist of AS edges inferred from traceroutes.Let the coverage of a set of measurements from $S \subset V'$ be:

$$\text{Cov}(S) = |\hat{E}| \tag{6.1}$$

Figure 6.2 illustrates one such prefix-based graph. The blue/hashed nodes indicate networks containing vantage points (*e.g.,* RIPE Atlas probes) and the purple/striped nodes indicate nodes that are discovered via traceroutes and single-homed customers of networks containing vantage points. Here $\hat{V}$ is the set of shaded nodes and $\hat{E}$ are the edges that connect them. White nodes represent nodes that we cannot discover via measurements towards P.

This leads us to the following problem definition for exploring the largest portion of the AS topology (G) with a fixed measurement budget of k traceroutes:

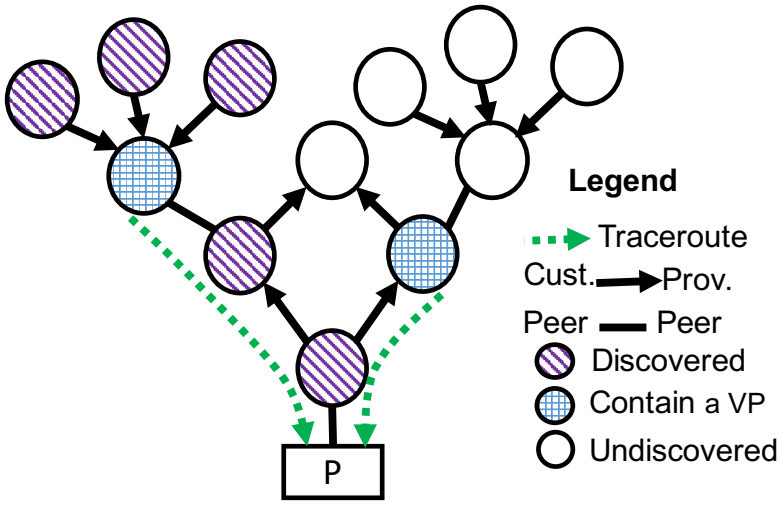


Figure 6.2: Example of a prefix-based DAG.

Input: Graph $G = (V, E)$ with vantage points V'.

$S \leftarrow \emptyset$
for ($i = 1$ to k):
    $S \leftarrow S + \text{argmax}_{s \in V'}\{\text{Cov}(S + s)\}$
return S

Figure 6.3: Greedy Vantage Point Selection

Problem 2 (PREFIX-COVER$_k$)**.** *Find $S \subset V'$, where $|S| \leq k$, that maximizes* Cov(S).

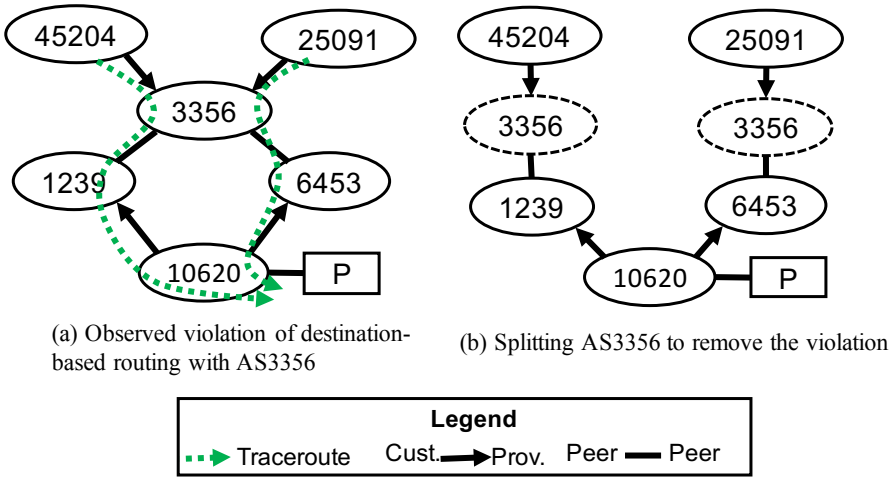
The PREFIX-COVER$_k$ problem is reducible to the MAX-COVER$_k$ problem in which the input is a number k and a collection $\mathcal{A} = \{A_1, A_2, \ldots\}$ of sets and the goal is to select a subset $\mathcal{A}' \subset \mathcal{A}$ of k sets that maximizes $|\cup_{A \in \mathcal{A}'} A|$. The reduction is immediate since each vantage point measurement can be thought of as a set A_i, and our goal is to maximize the coverage of edges by choosing k of these sets. MAX-COVER$_k$ is unfortunately NP-Hard. A well-known greedy algorithm provides the best possible approximation with a factor of $(1 - 1/e) \approx 0.63$ of the optimal solution [62]. However, since PREFIX-COVER$_k$ is a special case of MAX-COVER$_k$, where the G is largely expected to be a tree, we find that it can be solved exactly which we discuss in the next section.

6.3.3 Optimality for Destination-based Routing

We now discuss how a greedy algorithm to select vantage points (Algorithm 6.3) yields the optimal solution to PREFIX-COVER$_k$ when the graph G is a tree.

Theorem 57. *Algorithm 6.3 solves* PREFIX-COVER$_k$ *exactly when G is a tree, i.e., when there are no violations of destination-based routing.*

Proof. Let $S = \{s_1, s_2, \ldots, s_k\}$ denote the subset of vantage points returned by the greedy algorithm on G, where s_1 denotes the first vantage point chosen, s_2, denotes the second,



(a) Observed violation of destination-based routing with AS3356

(b) Splitting AS3356 to remove the violation

Figure 6.4: Example of violation of destination-based routing. Depending on the prior hop AS 3356 (Level 3) selects a different next-hop towards the destination. Splitting this node produces a tree-structured prefix DAG.

and so on. Recall that $\text{Cov}(S)$ denotes the coverage on G of the measurements run from S, and let $\mathcal{C}(S)$ denote the corresponding set of edges that are discovered. For the sake of contradiction assume that for some $i < k$, there is an optimal solution T such that $s_1, \ldots, s_{i-1} \in T$ but no optimal solution contains $s_1, \ldots, s_i$. Let

$$T = \{s_1, s_2, \ldots, s_{i-1}, t_i, t_{i+1}, \ldots t_k\}$$

where $s_i \neq t_j$ for all $j \geq i$. In other words, we assume that for some i, the first $i-1$ vantage points in S are also in T, but that s_i does not appear in T. For any set of vantage points A, define

$$\mathcal{C}'(A) = \mathcal{C}(A) \setminus \mathcal{C}(s_1, s_2, \ldots, s_{i-1})$$

and $\text{Cov}'(A) = |\mathcal{C}'(A)|$.

For any $j \geq i$, since s_i was chosen by the greedy algorithm before t_j we can infer that

$$\text{Cov}'(s_i) \geq \text{Cov}'(t_j) \ .$$

Let $j' \geq i$ be chosen to maximize $|\mathcal{C}'(s_i) \cap \mathcal{C}'(t_{j'})|$ and define $T' = (T \cup \{s_i\}) \setminus \{t_{j'}\}$. Observe that

$$\text{Cov}'(T') \geq \text{Cov}'(T) + \text{Cov}'(s_i) - \text{Cov}'(t_{j'})$$

Since $\text{Cov}'(s_i) \geq \text{Cov}'(t_{j'})$, we deduce that $\text{Cov}(T') \geq \text{Cov}(T)$ and so T' is also optimal. But $s_1, \ldots, s_i \in T'$ which is a contradiction to the assumption that no optimal solution contains $\{s_1, \ldots, s_i\}$. □

6.3.4 Prior-hop Violations of Destination-Based Routing.

When merging multiple traceroute-derived AS paths we observe cases that violate destination-based routing. Figure 6.4 shows one such example. Here, we observe AS 3356 (Level 3) selecting different next-hop ASes towards the same prefix, depending on the prior hop in the path[1]. In this case, it appears Level 3's routing decision is impacted by the prior

[1]To exclude the effect of churn in network paths, these traceroutes were run only a few minutes apart.

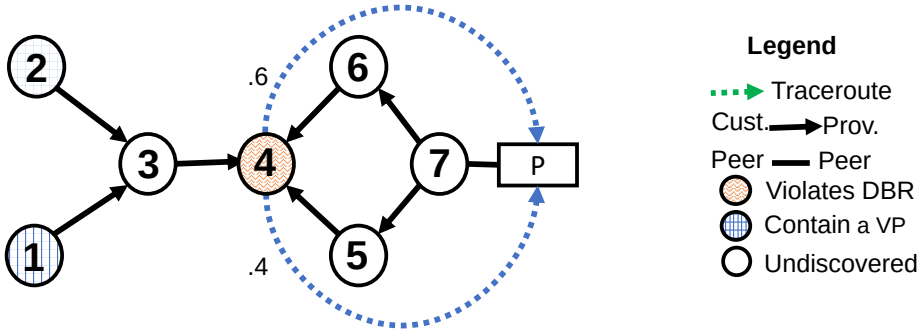


Figure 6.5: Traceroutes from vantage points are randomly routed from AS 4 independently among the two outgoing links according to the marked probabilities.

AS hop, thus we can "split" the node into two nodes, each of which represents Level 3's routing behavior for each of the prior hops. The resulting graph is now a tree.

In general, we can split AS nodes that violate destination-based routing based on their prior AS hop by creating a copy of the node for each prior hop, and adding to each copied node all outgoing edges associated with that prior hop.

The result of this process is a tree with the same number of edges as the original graph. We can run our greedy algorithm on this tree and optimize discovered coverage as before.

6.3.5 Other violations of destination-based routing.

Not all violations of destination-based routing are based on the prior AS hop. In these cases, it is often difficult to determine exactly what rule underlies the routing behavior and we simply treat this behavior as a random process. achieves an approximation factor of at least $(1 - 1/e)^2$.

Not all violations of destination-based routing are based on the prior AS hop. In these cases, it is often difficult to determine exactly what rule underlies the routing behavior and we simply treat this behavior as a random process. Figure 6.5 shows an example where traceroutes passing through node 4 are randomly routed on the left link with probability 0.4 and routed on the right link otherwise.

This gives rise to the problem of maximizing the *expected* coverage through our choice of vantage points. The set of edges covered by each vantage point v_i is now a random variable X_{v_i} whose value is the edges traversed by a random walk beginning at v_i and ending at P, where each step is chosen from the outgoing edges according to their routing probability. The problem we want to solve is:

Problem 3 (STOC-PREFIX-COVER$_k$)**.** *Find $S \subset V'$ with $|S| \leq k$, that maximizes $\mathbb{E}\left[|\cup_{v \in S} X_v|\right]$.*

For example, in Figure 6.5, X_1 is determined by starting at 1, continuing to 3 and 4, and randomly choosing either 5 with probability 0.4 or 6 with probability 0.6. Say 5 is chosen. Then continuing to 7 and ending at P yields the set of edges $X_1 = \{e_{1,3}, e_{3,4}, e_{4,5}, e_{5,7}, e_{7,P}\}$.

Stochastic Maximum Coverage. The above problem is special case of the *stochastic maximum coverage* problem. STOC-MAX-COVER$_k$ is a variant of MAX-COVER$_k$ where the input is an integer k and $\mathcal{A} = \{A_1, A_2, \ldots\}$ where A_i is a random set chosen according to some known distribution p_i. The goal is pick $\mathcal{A}' \subset \mathcal{A}$ to maximize $\mathbb{E}\left[|\cup_{A \in \mathcal{A}'} A|\right]$.

It can be shown that a natural extension of the greedy algorithm that, at each step, picks the set with the largest expected increase in the coverage, achieves a $1 - 1/e$ approximation [13]. However, note that using this algorithm in our context requires us to be *adaptive*, i.e., we select a vantage point, perform a measurement from that vantage point, and see the result of this measurement before selecting the next vantage point. This is in contrast to a *non-adaptive* algorithm that must choose the full set of k vantage points before running any measurements. There is a tradeoff between these two approaches: on the one hand, an adaptive approach may provide a solution of strictly better quality than a nonadaptive algorithm since it has strictly more information.

On the other hand, the adaptive algorithm is slower since after the adaptive algorithm chooses each measurement, it must wait for the traceroute to finish before computing the next measurement. This serial computation and measurement requirement prevents parallelization. Contrast this with the nonadaptive algorithm, which can immediately generate a complete schedule of measurements without actually performing any traceroutes. This schedule can be used to perform the actual traceroutes at any time/in parallel. This flexibility makes the nonadaptive algorithm desirable in some cases despite its strictly worse solution quality. For these reasons, we implement both versions and evaluate their solution quality theoretically and experimentally. As mentioned above, the adaptive greedy algorithm is guaranteed to achieve an approximation factor of at least $1-1/e$. The non-adaptive greedy algorithm is guaranteed an approximation factor of at least $(1 - 1/e)^2$; this follows from work by Asadpour and Nazerzadeh [13] on the more general STOC-MAX-COVER$_k$ problem.

6.3.6 A Note on Graph Coverage

The observant reader will note that the algorithmic focus here on graph coverage is related to the streaming coverage problems investigated in Chapter 3. However, the contexts in which graph problems are addressed in these two chapters are very different. PathCache has costly query access to the graph it is attempting to cover, while in Chapter 3 the input graph is only accessible as a stream. As a result, there is little technical overlap between the two problems.

6.4 Path Prediction

In this section, we outline how PathCache uses empirical data to predict paths. Section 6.4.1 explains how PathCache builds destination specific graphs and auxiliary tables to capture the routing behavior towards a prefix. Section 6.4.2 details how PathCache uses this information to define Markov chains for predicting paths between source and destination hosts on the Internet. Section 6.4.3 explains how simulations are used to fill in gaps in the Markov chains derived from traceroutes, enabling PathCache to predict paths for all queries to a measured prefix.

6.4.1 Constructing per-prefix DAGs

We gather a set of traceroutes publicly available on measurement platforms and those run in the topology discovery stage of PathCache. We aggregate traceroute hops into BGP

routed prefixes to make up the nodes in the DAGs. The process of converting IP paths to prefix-level (or AS-level) paths is involved and requires handling of complex corner cases. See the technical report [70] for more.

Let $\mathcal{P}$ be the set of discovered paths prefix-level paths derived from traceoutes. The process of constructing trusted per-destination graphs has two main components, generating the graph itself and computing the auxiliary "transition tables" that will be used to predict the sequence of edges that a path will take in the graph towards the destination prefix.

(1) **DAG Construction:** Take the union of all edges in the paths in $\mathcal{P}$ to form the directed acyclic graph $\mathcal{D} = (V, E)$.

(2) **Basic Transition Table:** For each edge $e \in E$, let c_e be the count of the number of paths including e.

Temporal transition tables and compression. We will be interested in predicting paths both based on the entirety of the observed data and based on data observed during a window of time. To support the latter, we need to augment our basic transition tables with an additional dimension. Let $t \in \{1, 2, \ldots, T\}$ index the relevant time period (*e.g.,* the last month or year) at the required resolution (*e.g.,* hours or days). Let $\mathcal{P}_t$ be the set of discovered paths at time t and let c_e^t be the number of paths in $\cup_{t' \geq t} \mathcal{P}_{t'}$ that includes e. By defining c_e^t in this way, note that $c_e^{t_2} - c_e^{t_1}$ is the number of paths in $\cup_{t_1 \leq t' < t_2} \mathcal{P}_{t'}$ that include e and this can be computed in $O(1)$ time rather than in $O(t_2 - t_1)$ time.

Unfortunately, storing $c_e^1, c_e^2, \ldots, e_e^T$ rather than just c_e increases the space to store the tables by a factor T and this may be significant. To ameliorate this situation, note that c_e^t is monotonically decreasing with t and hence it suffices to only store values for t where $c_e^t \neq c_e^{t-1}$ as the other values can be inferred from this information. By trading-off a small amount of accuracy we can further reduce the space as follows. Suppose we are willing to tolerate a $1 + \epsilon$ factor error in the values of c_e^t. Then, round each c_e^t to the nearest power of $1 + \epsilon$ and let $\tilde{c}_e^t$ be the resulting value. Then only storing $\tilde{c}_e^t$ for values of t where $\tilde{c}_e^t \neq \tilde{c}_e^{t-1}$ allows every c_e^t to be estimated up to a factor $1 + \epsilon$ whilst only storing at most $1 + \log_{1+\epsilon}(c_e^1)$ different values.

6.4.2 Path Prediction via Markov Chains

Learning transition probabilities. The graph for a given prefix P derived from traceroutes towards P defines the structure of the Markov chain PathCache uses for modeling the routing behavior towards the prefix. PathCache computes the transition probabilities of each Markov chain using Maximum Likelihood Estimation (MLE), given the traceroute counts stored in edge transition tables. With each edge $e \in E$ we assign a probability $p_e = c_e / \sum_{f \in N_u} c_f$ where N_u is the set of outgoing edges of u. If the user wishes to only train using traceroutes in a time window $\{t_1, t_1 + 1, \ldots, t_2 - 1\}$ we set $p_e = (c_e^{t_1} - c_e^{t_2}) / \sum_{f \in N_u} (c_f^{t_1} - c_f^{t_2})$

Once the Markov models are trained, PathCache obtains one Markov chain per destination prefix, where the states of the Markov chain represent BGP prefixes and the end state is the destination prefix P. Edges between the states are evidence of traffic towards P traversing them and the edge transition probabilities define the likelihood of traffic traversing

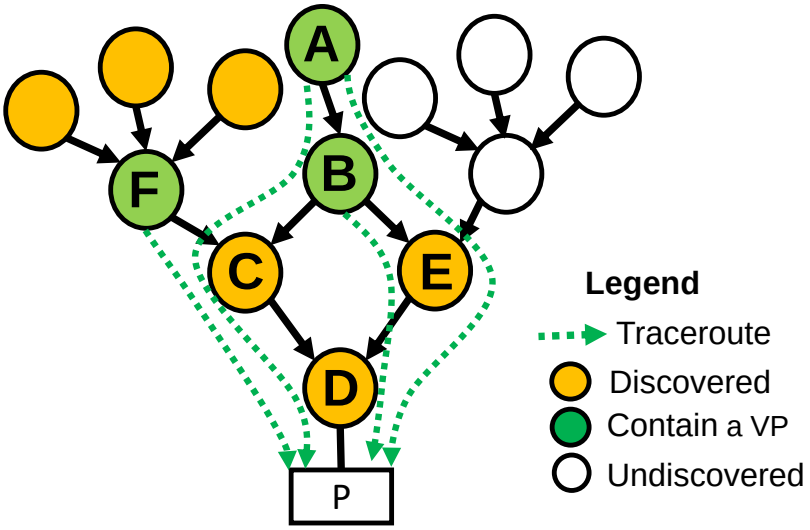


Figure 6.6: A DAG constructed from trusted traceroutes. Vantage point A sends two traceroutes which follow paths $ABCD$ and $ABED$. B sends one traceroute with path BED and F sends one traceroute with path FCD.

that edge. An example Markov chain is shown in Figure 6.6. In this chain A, B, C, and D are prefixes. We want to calculate the probability that a packet sent from source prefix A to destination prefix D follows the path $A \to B \to C \to D$.

Defining path probabilities. PathCache assumes probabilistic routing obeys a first-order Markov property. In other words, the probability of a packet choosing a next hop towards destination P only depends on the current hop. This dependence on current hops rather than all prior hops is inspired by the prevalence of next-hop routing on the Internet [73]. For the example in Figure 6.6, this first-order Markov assumption allows us to express the probability of the specified path as

$$\Pr(A, B, C, D) = \Pr(A) \cdot \Pr(B|A) \cdot \Pr(C|B) \cdot \Pr(D|C)$$

Since we have specified that the path begins at A, $\Pr(A) = 1$. In this case, $\Pr(B|A) = 1$ since all traceroutes originating from A go to B. $\Pr(C|B) = 1/3$ since there are 3 traceroutes that go through B, and one of them has next hop C. $\Pr(D|C) = 1$ since both of the traceroutes that go through C have a next hop of D. So $\Pr(A, B, C, D) = 1/3$.

Inferring most likely sequence of states. A naive way of predicting paths from prefix Markov chains would enumerate all paths from the source node S to the destination prefix D and return the one with the highest probability. However, a graph can have an exponential number of paths between the source and the destination, making this approach prohibitively expensive. A more efficient approach is as follows: for each edge $e = (u, v)$ define the length $\ell_e = -\log(p_e)$. This weight is always nonnegative and will be high if $p_e = \Pr(v|u)$ is low and vice versa. Furthermore, the probability of a path is inversely proportion to its length. We then find r shortest paths using Yen's algorithm [149] where the length of e is set to ℓ_e; these correspond to the r paths with the highest probability. We set $r = 5$ in PathCache's current implementation. PathCache returns a ranked list of these paths and their respective probabilities in response to a path query.

6.4.3 Splicing Empirical and Simulated Paths

If PathCache is faced with a query between a source AS and a destination prefix such that the source AS is not a state in the Markov chain of the destination prefix, it cannot predict the path using the chain alone. In such situations we query BGPSim for a policy compliant path from the source AS to the destination prefix. We keep each hop of this path, starting from the source ASN, until we reach an ASN that is present in the Markov chain for destination prefix. We predict the path between the ASN at the splice point and the destination prefix using methods described in the previous subsection. In this manner, PathCache can still return probability-ranked paths for such a query by considering the BGPSim splice of the path as fixed (with transition probabilities of 1).

6.5 Summary of Evaluation

Here we briefly summarize the findings of PathCache's experimental evaluation. See the technical report [70] for more.

- When compared to other strategies such as randomly choosing vantage points, or selecting vantage points from as many countries as possible, our measurement selection algorithm discovers 4 times the number of ASes using the same number of measurements ($k = 500$). These results are robust across different types of destination prefixes and across measurment platforms.
- To evaluate the accuracy of PathCache's path prediction algorithm, we trained its Markov model on all RIPE Atlas traceroutes from December 25, 2018 until March 4, 2019: approximately 4.5 billion traceroutes which allow us to learn the routing behavior towards all $\approx 500,000$ routable prefixes on the Internet. We then had PathCache predict paths from a source to a destination prefix, and compared these predicted paths with actual measured paths.
- 75% of PathCache's predicted paths differ from the corresponding measured path by at most 1 hop.
- When PathCache returned the correct path as one of its predictions, 90% of the time it was the path that PathCache assigned the highest probability to.
- By splicing empirical and simulated paths, PathCache is capable of responding to any path query, in contrast to iPlane and Sybil which only are able to respond to 70% of path queries.
- PathCache outperforms iPlane and Sybil in terms of path accuracy based on 2 metrics: edit distance and Jaccard index. The gap in performance increases with the length of the true path.
- We discovered that PathCache's per-destination directed graphs occasionally contain cycles. We discuss the implications of this in Section 6.7.

6.6 Case Studies

We are releasing the entire codebase of PathCache[2]. More importantly, we have deployed PathCache in beta version at `https://www.davidtench.com/deeplinks/pathcache`. PathCache can be queried for paths, either on the website or via the REST API[3]. The REST API can be incorporated programmatically into other systems.

In this section we demonstrate the impact of PathCache on real-world applications.

PathCache for Refraction Networking (RN). Refraction networking [147] is a recent technique for Internet censorship circumvention that incorporates the circumvention infrastructure into routers on the Internet. This technique has been considered more resilient to blocking by censors since it is hard to block individual routers on the Internet, while blocking source and destination of packets is relatively easy.

A key problem faced by Refraction Routing deployments today [66] is to place refraction routers in large ISPs such that client traffic gets intercepted by them. If client traffic follows a path without the refraction router on it, it leads to the failure of the refraction routing session.

We worked with the largest ISP-scale deployment of refraction routing [66] to use PathCache for predicting if a client refraction routing session will be successful. To predict a successful refraction routing connection, we use PathCache to predict the path between the client and refraction router. If the PathCache predicted path crosses specific prefixes within the deploying ISP, we conclude that the connection went via a refraction router and was successful (except for other non-networking failures). We find that in the current deployment of clients, PathCache can predict the prefix hop inside the deploying ISP which client traffic took when a RN session was successful, 100% of the time. In future, we are working towards incorporating the PathCache API into RN software for improved client performance.

PathCache to defend against routing adversaries. Researchers have found that anonymous communication via Tor [51] is susceptible to network-level adversaries launching routing attacks [145]. Several defenses against such attacks have been proposed that aim to avoid Tier-1 providers [18], use simulated BGP paths to avoid network-level adversaries [143], etc. PathCache's AS-level path prediction is highly accurate and readily available as a REST API which can be incorporated into Tor client software for defending against network-level adversaries.

6.7 Discussion and Future Work

In addition to their utility for path prediction, PathCache prefix graphs capture routing behavior in a novel way. We believe they can be used to revisit several classical problems in inter-domain routing. For instance:

BGP atoms. The networking research community has long studied the right granularity for modeling routing behavior on the Internet. One proposal is to find a set of routers that

[2]Link/reference omitted to preserve anonymity while the associated conference paper is still in submission.

[3]see the documentation on the website for details.

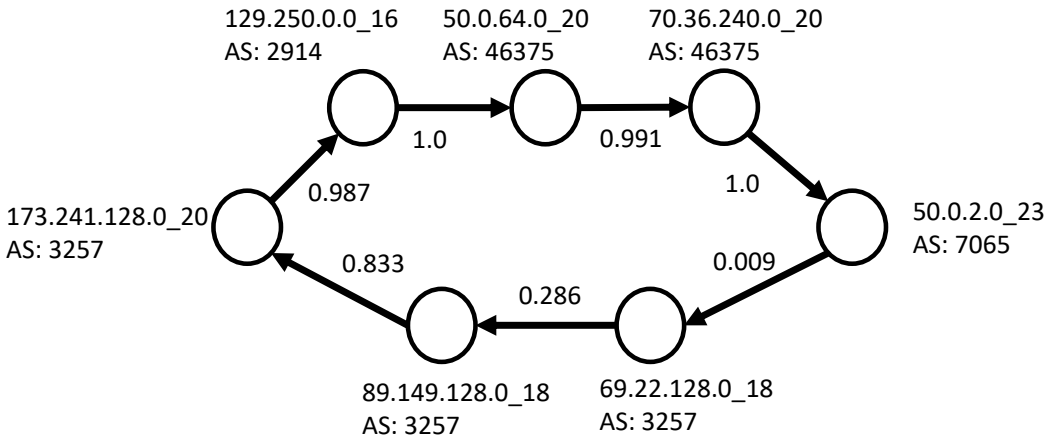


Figure 6.7: A cycle observed in the PathCache routing model of prefix 122.10.0.0/19. This cycle is across 4 ASes and lasted for 3 hours, as measured by traceroutes. Node ASNs, prefixes and edge probabilities are annotated.

route towards the Internet similarly, called BGP atoms [3]. We note that since PathCache has a view of the routing behavior of all prefixes on the Internet, using measures of graph similarity across prefixes, PathCache's Markov chains can potentially provide a way to infer BGP atoms.

Analyzing routing convergence. In Section 6.5 we described the existence of cycles in PathCache's per-destination graphs. While these cycles are rare across ASes and have very short duration, we think they offer a new perspective on the analysis of BGP route convergence. Figure 6.7 shows one such cycle observed in the PathCache graph for prefix 122.10.0.0/19 which lasted for 3 hours. Combining PathCache's data plane analysis with BGP announcements can help us identify the cause of these cycles and how long it took for them to be resolved in the control as well as the data plane.

CHAPTER 7

CONCLUSION

We conclude by briefly summarizing the work presented in this thesis, and discuss opportunities for future research in these areas.

Connectivity in Dynamic (Hyper-)Graph Streams. Prior to this work the main success story in dynamic streaming graph connectivity had been in computing edge connectivity. Vertex connectivity exhibits markedly different combinatorial structure than edge connectivity and appeared to be more difficult in the dynamic grpah stream model. In Chapter 2, we presented the first linear sketches for estimating vertex connectivity and constructing hypergraph sparsifiers in dynamic graph streams, We also extended a graph reconstruction result to a larger class of graphs.

Coverage in Data Streams. In Chapter 3, we presented a variety of efficient algorithms for computing Max-k-Cover and Max-k-UniqueCover in the data stream model. These problems are closely related to a range of important graph problems including matching, partial vertex cover, and capacitated maximum cut. Our results improve upon the state of the art for Max-k-Cover streaming algorithms, and are the first of their kind for the Max-k-UniqueCover problem.

Temporal Graph Streaming. In Chapter 4, we initiated the study of temporal graph algorithms in the streaming setting. We provide sketching algorithms for some notions of temporal connectivity, and prove a set of strong lower bounds for others. This work is the first step in a more complete investigation of these rich combinatorial structures in the streaming setting.

MESH. Chapter 5 introduced MESH, a memory allocator that efficiently performs *compaction without relocation* to save memory for unmanaged languages. We show analytically that MESH provably avoids catastrophic memory fragmentation with high probability, and empirically show that MESH can substantially reduce memory fragmentation for memory-intensive applications written in C/C++ with low runtime overhead.

PathCache. Chapter 6 introduced PathCache, a system that predicts network paths between arbitrary hosts on the Internet using historical knowledge of the control and data plane. PathCache's strategy for exploring network paths discovers 4 times more AS hops than the current state of the art methods. The PathCache system is capable of accurately

predicting destination-based Internet paths at scale. We investigated the utility of PathCache in improving real-world applications for circumventing Internet censorship and preserving anonymity online.

7.1 Future Work

Temporal graph streams have never before been studied and many fundamental algorithmic questions have yet to be answered. Does Conjecture 47 hold, implying that even relatively basic temporal reachability problems are difficult to compute in streams? If so, are there other notions of reachability which are still useful but require less space to compute? If instead these reachability problems are feasible, how might we compute more complicated connectivity problems, such as edge or vertex connectivity? What other important temporal graph properties are computable in the streaming setting? Can we use sketching to create fingerprints of temporal graph streams, such that any two streams which define the same temporal graph have the same fingerprint, and a different temporal graph has a different fingerprint with high probability? Is it possible to determine in the streaming setting whether T contains a journey that visits every node at least once? Can we sparsify a temporal graph while maintaining pairwise reachability? Can we perform this sparsification if distance (or journey duration) must be approximately maintained?

In Chapter 6, we described the existence of cycles in PathCache's per-destination graphs, a phenomenon which violates the common assumption of that Internet routing is destination-based. Are these apparent violations merely artifacts of vantage point measurement capabilities or are they evidence of poorly understood BGP routing dynamics? Answering this question requires characterizing frequency, duration, length, and other properties of these cycles which is an algorithmic challenge at Internet scale.

Memory systems are described by a hierarchy of transfer block size and latency. The classical assumption is that the blocks and latencies grow exponentially as one moves down the memory hierarchy. Cache is larger and slower than register, RAM is larger and slower than cache, and disk is larger and slower than RAM. This size and accessibility gradient motivates both external memory data structures and streaming algorithms. Recently, the hierarchy has been flattening. For instance, random I/O bandwidth in RAM is roughly comparable to sequential I/O bandwidth to new NVMe devices [43]. This new storage landscape has not yet been investigated thoroughly, but early work [132] suggests that techniques from the data stream model will prove useful in designing data structures that are optimized for the tradeoffs of modern external memory. The existence of this new external memory hardware motivates new models for the streaming domain as well. Which streaming problems become easier if, for example, the algorithm is allowed a small amount of random access to the input in addition to the stream? What if it has access to its own sequentially accessible writeable memory, which is asymptotically larger than its random access memory but still asymptotically smaller than the size of the stream?

BIBLIOGRAPHY

[1] Ablayev, Farid M. Lower bounds for one-way probabilistic communication complexity and their application to space complexity. *Theor. Comput. Sci. 157*, 2 (1996), 139–159.

[2] Adger, Mads. *[Blink-dev] Oilpan - experimenting with a garbage collected heap for the DOM*, 2013. `http://bit.ly/2pwDhwk`.

[3] Afek, Yehuda, Ben-Shalom, Omer, and Bremler-Barr, Anat. On the structure and application of bgp policy atoms. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment* (New York, NY, USA, 2002), IMW '02, ACM, pp. 209–214.

[4] Ageev, Alexander A., and Sviridenko, Maxim. Pipage rounding: A new method of constructing algorithms with proven performance guarantee. *J. Comb. Optim. 8*, 3 (2004), 307–328.

[5] Ahn, Kook Jin, and Guha, Sudipto. Linear programming in the semi-streaming model with application to the maximum matching problem. *Inf. Comput. 222* (2013), 59–79.

[6] Ahn, Kook Jin, Guha, Sudipto, and McGregor, Andrew. Analyzing graph structure via linear measurements. In *Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012* (2012), pp. 459–467.

[7] Ahn, Kook Jin, Guha, Sudipto, and McGregor, Andrew. Graph sketches: sparsification, spanners, and subgraphs. In *31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (2012), pp. 5–14.

[8] Ahn, Kook Jin, Guha, Sudipto, and McGregor, Andrew. Spectral sparsification in dynamic graph streams. In *APPROX* (2013), pp. 1–10.

[9] Alaluf, Naor, Ene, Alina, Feldman, Moran, Nguyen, Huy L., and Suh, Andrew. Optimal streaming algorithms for submodular maximization with cardinality constraints. In *ICALP* (2020), vol. 168 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 6:1–6:19.

[10] Alweiss, Ryan, Lovett, Shachar, Wu, Kewen, and Zhang, Jiapeng. Improved bounds for the sunflower lemma. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020* (2020), pp. 624–630.

[11] Anagnostopoulos, Aris, Becchetti, Luca, Bordino, Ilaria, Leonardi, Stefano, Mele, Ida, and Sankowski, Piotr. Stochastic query covering for fast approximate document retrieval. *ACM Trans. Inf. Syst. 33*, 3 (2015), 11:1–11:35.

[12] Anwar, R., Niaz, H., Choffnes, D., Cunha, I., Gill, P., and Katz-Bassett, E. Investigating interdomain routing policies in the wild. In *ACM IMC* (2015).

[13] Asadpour, Arash, and Nazerzadeh, Hamid. Maximizing stochastic monotone submodular functions. *Management Science 62*, 8 (2016), 2374–2391.

[14] Assadi, Sepehr. Tight space-approximation tradeoff for the multi-pass streaming set cover problem. In *PODS* (2017), ACM, pp. 321–335.

[15] Assadi, Sepehr, Khanna, Sanjeev, and Li, Yang. Tight bounds for single-pass streaming complexity of the set cover problem. In *STOC* (2016), ACM, pp. 698–711.

[16] Badanidiyuru, Ashwinkumar, Mirzasoleiman, Baharan, Karbasi, Amin, and Krause, Andreas. Streaming submodular maximization: massive data summarization on the fly. In *KDD* (2014), ACM, pp. 671–680.

[17] Bar-Yossef, Ziv, Jayram, T. S., Kumar, Ravi, Sivakumar, D., and Trevisan, Luca. Counting distinct elements in a data stream. In *RANDOM* (2002), vol. 2483 of *Lecture Notes in Computer Science*, Springer, pp. 1–10.

[18] Barton, Armon, and Wright, Matthew. Denasa: Destination-naive as-awareness in anonymous communications. *Proceedings on Privacy Enhancing Technologies 2016*, 4 (2016), 356–372.

[19] Becker, Florent, Matamala, Martín, Nisse, Nicolas, Rapaport, Ivan, Suchan, Karol, and Todinca, Ioan. Adding a referee to an interconnection network: What can(not) be computed in one round. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011* (2011), pp. 508–514.

[20] Benczúr, András A., and Karger, David R. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *STOC* (1996), pp. 47–55.

[21] Berger, Emery D., McKinley, Kathryn S., Blumofe, Robert D., and Wilson, Paul R. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2000), ACM Press, pp. 117–128.

[22] Berger, Emery D., and Zorn, Benjamin G. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)* (New York, NY, USA, 2006), ACM Press, pp. 158–168.

[23] Berman, Kenneth A. Vulnerability of scheduled networks and a generalization of menger's theorem. *Networks 28*, 3 (1996), 125–134.

[24] Bhadra, Sandeep, and Ferreira, Afonso. Complexity of connected components in evolving graphs and the computation of multicast trees in dynamic networks. vol. 3, pp. 259–270.

[25] Bonnet, Édouard, Paschos, Vangelis Th., and Sikora, Florian. Parameterized exact and approximation algorithms for maximum k-set cover and related satisfiability problems. *RAIRO Theor. Informatics Appl. 50*, 3 (2016), 227–240.

[26] Braverman, Vladimir, Ostrovsky, Rafail, and Vilenchik, Dan. How hard is counting triangles in the streaming model? In *ICALP (1)* (2013), vol. 7965 of *Lecture Notes in Computer Science*, Springer, pp. 244–254.

[27] Bury, Marc, and Schwiegelshohn, Chris. Sublinear estimation of weighted matchings in dynamic data streams. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings* (2015), pp. 263–274.

[28] Çatalyürek, Ümit V., and Aykanat, Cevdet. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst. 10*, 7 (1999), 673–693.

[29] Çatalyürek, Ümit V., Boman, Erik G., Devine, Karen D., Bozdag, Doruk, Heaphy, Robert T., and Riesen, Lee Ann. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput. 69*, 8 (2009), 711–724.

[30] Censor-Hillel, Keren, Ghaffari, Mohsen, Giakkoupis, George, Haeupler, Bernhard, and Kuhn, Fabian. Tight bounds on vertex connectivity under vertex sampling. In *ACM-SIAM Symposium on Discrete Algorithms, SODA 2015* (2015).

[31] Censor-Hillel, Keren, Ghaffari, Mohsen, and Kuhn, Fabian. A new perspective on vertex connectivity. In *Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014* (2014), pp. 546–561.

[32] Chakrabarti, Amit, and Kale, Sagar. Submodular maximization meets streaming: matchings, matroids, and more. *Math. Program. 154*, 1-2 (2015), 225–247.

[33] Chakrabarti, Amit, Khot, Subhash, and Sun, Xiaodong. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *IEEE Conference on Computational Complexity* (2003), IEEE Computer Society, pp. 107–117.

[34] Chakrabarti, Amit, and Wirth, Anthony. Incidence geometries and the pass complexity of semi-streaming set cover. In *SODA* (2016), SIAM, pp. 1365–1373.

[35] Chekuri, Chandra, Gupta, Shalmoli, and Quanrud, Kent. Streaming algorithms for submodular function maximization. In *ICALP (1)* (2015), vol. 9134 of *Lecture Notes in Computer Science*, Springer, pp. 318–330.

[36] Cheriyan, J., Kao, M. Y., and Thurimella, R. Scan-first search and sparse certificates: an improved parallel algorithm for k-vertex connectivity. *SIAM Journal on Computing 22*, 1 (1993), 157–174.

[37] Chitnis, Rajesh, and Cormode, Graham. Towards a theory of parameterized streaming algorithms. In *14th International Symposium on Parameterized and Exact Computation, IPEC 2019, September 11-13, 2019, Munich, Germany* (2019), pp. 7:1–7:15.

[38] Chitnis, Rajesh, Cormode, Graham, Esfandiari, Hossein, Hajiaghayi, MohammadTaghi, McGregor, Andrew, Monemizadeh, Morteza, and Vorotnikova, Sofya. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *SODA* (2016), SIAM, pp. 1326–1344.

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

[40] Chitnis, Rajesh Hemant, Cormode, Graham, Hajiaghayi, Mohammad Taghi, and Monemizadeh, Morteza. Parameterized streaming: Maximal matching and vertex cover. In *SODA* (2015), SIAM, pp. 1234–1251.

[41] Cohen, Nachshon, and Petrank, Erez. Limitations of partial compaction: Towards practical bounds. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 309–320.

[42] Cohen, Nachshon, and Petrank, Erez. Limitations of partial compaction: Towards practical bounds. *ACM Trans. Program. Lang. Syst. 39*, 1 (Mar. 2017), 2:1–2:44.

[43] Conway, Alex, Gupta, Abhishek, Tai, Amy, Spillane, Richard, Chidabaram, Vijay, Farach-Colton, Martin, and Johnson, Rob. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *USENIX ATC 2020* (2020).

[44] Coppeard, Jon. *Compacting Garbage Collection in SpiderMonkey*, 2015. `https://mzl.la/2rntQlY`.

[45] Cormode, Graham, Datar, Mayur, Indyk, Piotr, and Muthukrishnan, S. Comparing data streams using hamming norms (how to zero in). *IEEE Trans. Knowl. Data Eng. 15*, 3 (2003), 529–540.

[46] Cormode, Graham, and Muthukrishnan, S. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms 55*, 1 (2005), 58–75.

[47] Crouch, Michael, and Stubbs, Daniel S. Improved streaming algorithms for weighted matching, via unweighted matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain* (2014), pp. 96–104.

[48] Crouch, Michael S., McGregor, Andrew, and Stubbs, Daniel. Dynamic graphs in the sliding-window model. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings* (2013), pp. 337–348.

[49] Demaine, Erik D., Feige, Uriel, Hajiaghayi, MohammadTaghi, and Salavatipour, Mohammad R. Combination can be hard: Approximability of the unique coverage problem. *SIAM J. Comput. 38*, 4 (2008), 1464–1483.

[50] Detlefs, David. Garbage collection and run-time typing as a C++ library. In *Proceedings of the 1992 USENIX C++ Conference* (1992), USENIX Association, pp. 37–56.

[51] Dingledine, Roger, Mathewson, Nick, and Syverson, Paul. Tor: The second-generation onion router. Tech. rep., Naval Research Lab Washington DC, 2004.

[52] Dom, Michael, Guo, Jiong, Niedermeier, Rolf, and Wernicke, Sebastian. Minimum membership set covering and the consecutive ones property. In *SWAT* (2006), vol. 4059 of *Lecture Notes in Computer Science*, Springer, pp. 339–350.

[53] Edelson, Daniel R. Precompiling C++ for garbage collection. In *Proceedings of the International Workshop on Memory Management* (London, UK, UK, 1992), IWMM '92, Springer-Verlag, pp. 299–314.

[54] Emek, Yuval, and Rosén, Adi. Semi-streaming set cover - (extended abstract). In *ICALP* (2014), pp. 453–464.

[55] Emek, Yuval, and Rosén, Adi. Semi-streaming set cover. *ACM Trans. Algorithms 13*, 1 (2016), 6:1–6:22.

[56] Eppstein, David, Galil, Zvi, Italiano, Giuseppe F., and Nissenzweig, Amnon. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM 44*, 5 (1997), 669–696.

[57] Epstein, Leah, Levin, Asaf, Mestre, Julián, and Segev, Danny. Improved approximation guarantees for weighted matching in the semi-streaming model. *SIAM J. Discrete Math. 25*, 3 (2011), 1251–1265.

[58] Erlebach, Thomas, and van Leeuwen, Erik Jan. Approximating geometric coverage problems. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008* (2008), pp. 1267–1276.

[59] Evans, Jason. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada* (2006).

[60] Feige, U., Hajiyaghayi, M., and Lee, J. R. Improved approximation algorithms for minimum-weight vertex separators. *Proc. of STOC* (2005).

[61] Feige, Uriel. A threshold of ln n for approximating set cover. *J. ACM 45*, 4 (1998), 634–652.

[62] Feige, Uriel. A threshold of ln n for approximating set cover. *J. ACM 45*, 4 (July 1998), 634–652.

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

[64] Feldman, Moran, Norouzi-Fard, Ashkan, Svensson, Ola, and Zenklusen, Rico. The one-way communication complexity of submodular maximization with applications to streaming and robustness. In *STOC* (2020), ACM, pp. 1363–1374.

[65] Fenichel, Robert R., and Yochelson, Jerome C. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM 12*, 11 (Nov. 1969), 611–612.

[66] Frolov, Sergey, Douglas, Fred, Scott, Will, McDonald, Allison, VanderSloot, Benjamin, Hynes, Rod, Kruger, Adam, Kallitsis, Michalis, Robinson, David G, Schultze, Steve, et al. An isp-scale deployment of tapdance. In *7th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 17)* (2017).

[67] Fung, Wai Shing, Hariharan, Ramesh, Harvey, Nicholas J. A., and Panigrahi, Debmalya. A general framework for graph sparsification. In *STOC* (2011), pp. 71–80.

[68] Gaur, Daya Ram, Krishnamurti, Ramesh, and Kohli, Rajeev. Erratum to: The capacitated max *k*-cut problem. *Math. Program. 126*, 1 (2011), 191.

[69] Ghemawat, Sanjay, and Menage, Paul. TCMalloc: Thread-caching malloc, 2007. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[70] Gill, Phillipa, McGregor, Andrew, Singh, Rachee, and Tench, David. Pathcache: A network path prediction toolkit. `https://www.davidtench.com/deeplinks/pathcache_paper`.

[71] Gill, Phillipa, Schapira, Michael, and Goldberg, Sharon. Let the market drive deployment: A strategy for transitioning to bgp security. *SIGCOMM Comput. Commun. Rev. 41*, 4 (Aug. 2011), 14–25.

[72] Gill, Phillipa, Schapira, Michael, and Goldberg, Sharon. Modeling on quicksand: Dealing with the scarcity of ground truth in interdomain routing data. *ACM SIGCOMM Computer Communication Review 42*, 1 (2012), 40–46.

[73] Gill, Phillipa, Schapira, Michael, and Goldberg, Sharon. A survey of interdomain routing policies. *In ACM Computer Communications Review (CCR).* (Jan 2014).

[74] Goel, Ashish, Kapralov, Michael, and Khanna, Sanjeev. On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012* (2012), pp. 468–485.

[75] Goel, Ashish, Kapralov, Michael, and Post, Ian. Single pass sparsification in the streaming model with edge deletions. *CoRR abs/1203.4900* (2012).

[76] Guruswami, Venkatesan, and Onak, Krzysztof. Superlinear lower bounds for multipass graph processing. In *Proceedings of the 28th Conference on Computational Complexity, CCC 2013, Palo Alto, California, USA, 5-7 June, 2013* (2013), pp. 287–298.

[77] Hansen, Wilfred J. Compact list representation: Definition, garbage collection, and system implementation. *Commun. ACM 12*, 9 (Sept. 1969), 499–507.

[78] Har-Peled, Sariel, Indyk, Piotr, Mahabadi, Sepideh, and Vakilian, Ali. Towards tight bounds for the streaming set cover problem. In *PODS* (2016), ACM, pp. 371–383.

[79] Hara, Kentaro. *State of Blink's Speed*, Sept. 2017. `https://docs.google.com/presentation/d/1Az-F3CamBq6hZ5QqQt-ynQEMWEhHY1VTvlRwL7b_6TU`.

[80] Huang, Chien-Chung, Kakimura, Naonori, and Yoshida, Yuichi. Streaming algorithms for maximizing monotone submodular functions under a knapsack constraint. In *APPROX-RANDOM* (2017), vol. 81 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 11:1–11:14.

[81] Huang, Yuchi, Liu, Qingshan, and Metaxas, Dimitris N. Video object segmentation by hypergraph cut. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA* (2009), pp. 1738–1745.

[82] Indyk, Piotr, Mahabadi, Sepideh, Rubinfeld, Ronitt, Ullman, Jonathan, Vakilian, Ali, and Yodpinyanee, Anak. Fractional set cover in the streaming model. In *APPROX-RANDOM* (2017), vol. 81 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 12:1–12:20.

[83] Indyk, Piotr, and Vakilian, Ali. Tight trade-offs for the maximum k-coverage problem in the general streaming model. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), pp. 200–217.

[84] Ipek, Engin, Condit, Jeremy, Nightingale, Edmund B., Burger, Doug, and Moscibroda, Thomas. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. *SIGPLAN Not. 45*, 3 (Mar. 2010), 3–14.

[85] Ito, Takehiro, Nakano, Shin-Ichi, Okamoto, Yoshio, Otachi, Yota, Uehara, Ryuhei, Uno, Takeaki, and Uno, Yushi. A 4.31-approximation for the geometric unique coverage problem on unit disks. *Theor. Comput. Sci. 544* (2014), 14–31.

[86] Johnstone, Mark S., and Wilson, Paul R. The memory fragmentation problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management* (New York, NY, USA, 1998), ISMM '98, ACM, pp. 26–36.

[87] Jowhari, Hossein, Saglam, Mert, and Tardos, Gábor. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *PODS* (2011), pp. 49–58.

[88] Kallaugher, John, McGregor, Andrew, Price, Eric, and Vorotnikova, Sofya. The complexity of counting cycles in the adjacency list streaming model. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), pp. 119–133.

[89] Kapralov, Michael. Better bounds for matchings in the streaming model. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013* (2013), pp. 1679–1697.

[90] Kapralov, Michael, Khanna, Sanjeev, and Sudan, Madhu. Approximating matching size from random streams. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014* (2014), pp. 734–751.

[91] Kapralov, Michael, Khanna, Sanjeev, and Sudan, Madhu. Streaming lower bounds for approximating MAX-CUT. In *SODA* (2015), SIAM, pp. 1263–1282.

[92] Kapralov, Michael, Khanna, Sanjeev, Sudan, Madhu, and Velingker, Ameya. $(1 + \omega(1))$-approximation to MAX-CUT requires linear space. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19* (2017), pp. 1703–1722.

[93] Kapralov, Michael, Lee, Yin Tat, Musco, Cameron, Musco, Christopher, and Sidford, Aaron. Single pass spectral sparsification in dynamic streams. In *FOCS* (2014).

[94] Kapralov, Michael, and Woodruff, David P. Spanners and sparsifiers in dynamic streams. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014* (2014), pp. 272–281.

[95] Karger, David R. Random sampling in cut, flow, and network design problems. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing* (New York, NY, USA, 1994), ACM, pp. 648–657.

[96] Karger, David R. Random sampling in cut, flow, and network design problems. In *STOC* (1994), pp. 648–657.

[97] Katz-Bassett, E., Madhyastha, H., Adhikari, V., Scott, C., Sherry, J., van Wesep, P., Krishnamurthy, A., and Anderson, T. Reverse traceroute. In *USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2010).

[98] Katz-Bassett, Ethan, Marchetta, Pietro, Calder, Matt, Chiu, Yi-Ching, Cunha, Italo, Madhyastha, Harsha, and Giotsas, Vasileios. Sibyl: A practical internet route oracle. In *USENIX NSDI* (2016).

[99] Kempe, David, Kleinberg, Jon, and Kumar, Amit. Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences 64*, 4 (2002), 820 – 842.

[100] Kempe, David, Kleinberg, Jon M., and Tardos, Éva. Maximizing the spread of influence through a social network. *Theory of Computing 11* (2015), 105–147.

[101] Kogan, Dmitry, and Krauthgamer, Robert. Sketching cuts in graphs and hypergraphs. In *6th Innovations in Theoretical Computer Science* (2015).

[102] Konrad, Christian. Maximum matching in turnstile streams. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings* (2015), pp. 840–852.

[103] Konrad, Christian, Magniez, Frédéric, and Mathieu, Claire. Maximum matching in semi-streaming with few passes. In *APPROX-RANDOM* (2012), vol. 7408 of *Lecture Notes in Computer Science*, Springer, pp. 231–242.

[104] Konrad, Christian, and Rosén, Adi. Approximating semi-matchings in streaming and in two-party communication. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I* (2013), pp. 637–649.

[105] Krause, Andreas, and Guestrin, Carlos. Near-optimal observation selection using submodular functions. In *AAAI* (2007), AAAI Press, pp. 1650–1654.

[106] Kuhn, Fabian, von Rickenbach, Pascal, Wattenhofer, Roger, Welzl, Emo, and Zollinger, Aaron. Interference in cellular networks: The minimum membership set cover problem. In *COCOON* (2005), vol. 3595 of *Lecture Notes in Computer Science*, Springer, pp. 188–198.

[107] Kutzkov, Konstantin, and Pagh, Rasmus. Triangle counting in dynamic graph streams. In *Algorithm Theory - SWAT 2014 - 14th Scandinavian Symposium and Workshops, Copenhagen, Denmark, July 2-4, 2014. Proceedings* (2014), pp. 306–318.

[108] Li, Yi, Nguy˜ên, Huy L., and Woodruff, David P. Turnstile streaming algorithms might as well be linear sketches.

[109] Madhyastha, Harsha V., Isdal, Tomas, Piatek, Michael, Dixon, Colin, Anderson, Thomas, Krishnamurthy, Arvind, and Venkataramani, Arun. iPlane: An Information Plane for Distributed Services. In *Proc. of Operatings System Design and Implementation* (2006).

[110] Madhyastha, Harsha V, Katz-Bassett, Ethan, Anderson, Thomas E, Krishnamurthy, Arvind, and Venkataramani, Arun. iplane nano: Path prediction for peer-to-peer applications. In *NSDI* (2009), vol. 9, pp. 137–152.

[111] Manurangsi, Pasin. A note on max k-vertex cover: Faster fpt-as, smaller approximate kernel and improved approximation. In *2nd Symposium on Simplicity in Algorithms, SOSA@SODA 2019, January 8-9, 2019 - San Diego, CA, USA* (2019), pp. 15:1–15:21.

[112] Mariani, Rico. *Garbage Collector Basics and Performance Hints*, 2003. `https://msdn.microsoft.com/en-us/library/ms973837.aspx`.

[113] McGregor, Andrew. Finding graph matchings in data streams. *APPROX-RANDOM* (2005), 170–181.

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

[115] McGregor, Andrew, Rudra, Atri, and Uurtamo, Steve. Polynomial fitting of data streams with applications to codeword testing, 2011.

[116] McGregor, Andrew, and Vorotnikova, Sofya. Planar matching in streams revisited. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2016, September 7-9, 2016, Paris, France* (2016), pp. 17:1–17:12.

[117] McGregor, Andrew, and Vorotnikova, Sofya. Triangle and four cycle counting in the data stream model. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020* (2020), pp. 445–456.

[118] McGregor, Andrew, Vorotnikova, Sofya, and Vu, Hoa T. Better algorithms for counting triangles in data streams. In *PODS* (2016), ACM, pp. 401–411.

[119] McGregor, Andrew, and Vu, Hoa T. Better streaming algorithms for the maximum coverage problem. In *ICDT* (2017), vol. 68 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 22:1–22:18.

[120] McGregor, Andrew, and Vu, Hoa T. Better streaming algorithms for the maximum coverage problem. *Theory of Computing Systems* (2018), 1–25.

[121] Menger, Karl. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae 10*, 1 (1927), 96–115.

[122] Mertzios, George B., Michail, Othon, Chatzigiannakis, Ioannis, and Spirakis, Paul G. Temporal network optimization subject to connectivity constraints. In *Automata, Languages, and Programming* (Berlin, Heidelberg, 2013), Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, Eds., Springer Berlin Heidelberg, pp. 657–668.

[123] Michail, Othon. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics 12*, 4 (2016), 239–280.

[124] Michail, Othon, and Spirakis, Paul G. Traveling salesman problems in temporal graphs. *Theoretical Computer Science 634* (2016), 1 – 23.

[125] Microystems, Sun. Memory management in the java hotspot™ virtual machine, 2006.

[126] Misra, Neeldhara, Moser, Hannes, Raman, Venkatesh, Saurabh, Saket, and Sikdar, Somnath. The parameterized complexity of unique coverage and its variants. *Algorithmica 65*, 3 (2013), 517–544.

[127] Mühlbauer, Wolfgang, Uhlig, Steve, Fu, Bingjie, Meulle, Mickael, and Maennel, Olaf. In search for an appropriate granularity to model routing policies. *SIGCOMM Comput. Commun. Rev. 37*, 4 (Aug. 2007), 145–156.

[128] Nithyanand, Rishab, Singh, Rachee, Cho, Shinyoung, and Gill, Phillipa. Holding all the ases: Identifying and circumventing the pitfalls of as-aware tor client design. *CoRR abs/1605.03596* (2016).

[129] Norouzi-Fard, Ashkan, Tarnawski, Jakub, Mitrovic, Slobodan, Zandieh, Amir, Mousavifar, Aidasadat, and Svensson, Ola. Beyond 1/2-approximation for submodular maximization on massive data streams. In *ICML* (2018), vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 3826–3835.

[130] Novark, Gene, and Berger, Emery D. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 573–584.

[131] Novark, Gene, Berger, Emery D., and Zorn, Benjamin G. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2009)* (New York, NY, USA, 2009), ACM, pp. 397–407.

[132] Pandey, Prashant, Singh, Shikha, Bender, Michael A., Berry, Jonathan W., Farach-Colton, Martín, Johnson, Rob, Kroeger, Thomas M., and Phillips, Cynthia A. Timely reporting of heavy hitters using external memory. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 1431–1446.

[133] Radhakrishnan, Jaikumar, and Shannigrahi, Saswata. Streaming algorithms for 2-coloring uniform hypergraphs. In *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings* (2011), pp. 667–678.

[134] Rao, Anup. Coding for sunflowers. *CoRR abs/1909.04774* (2019).

[135] Rigby, Dave. *[jemalloc] expose hints that will allow applications to perform runtime active defragmentation Bug 566*, 2017. `https://github.com/jemalloc/jemalloc/issues/566`.

[136] RIPE Atlas. `https://atlas.ripe.net/`.

[137] Robson, J. M. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal 20*, 3 (1977), 242.

[138] Saha, Barna, and Getoor, Lise. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *SIAM International Conference on Data Mining, SDM 2009, April 30 - May 2, 2009, Sparks, Nevada, USA* (2009), pp. 697–708.

[139] Sanfilippo, Salvatore. Redis 4.0 ga is out!, 2017. `https://groups.google.com/forum/#!msg/redis-db/5Kh3viziYGQ/58TKLwX0AAAJ`.

[140] Sankowski, Piotr. Faster dynamic matchings and vertex connectivity. In *Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007* (2007), pp. 118–126.

[141] Schmidt, Jeanette P., Siegel, Alan, and Srinivasan, Aravind. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math. 8*, 2 (1995), 223–250.

[142] Schoenick, John. Are we slim yet?, 2017. `https://areweslimyet.com/faq.htm`.

[143] Starov, Oleksii, Nithyanand, Rishab, Zair, Adva, Gill, Phillipa, and Schapira, Michael. Measuring and mitigating as-level adversaries against tor. In *NDSS* (2016).

[144] Sun, He. Counting hypergraphs in data streams. *CoRR abs/1304.7456* (2013).

[145] Sun, Yixin, Edmundson, Anne, Vanbever, Laurent, Li, Oscar, Rexford, Jennifer, Chiang, Mung, and Mittal, Prateek. Raptor: Routing attacks on privacy in tor. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 271–286.

[146] Wustrow, Eric, Wolchok, Scott, Goldberg, Ian, and Halderman, J. Alex. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 30–30.

[147] Wustrow, Eric, Wolchok, Scott, Goldberg, Ian, and Halderman, J. Alex. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 30–30.

[148] Yamaguchi, Y., Ogawa, A., Takeda, A., and Iwata, S. Cyber security analysis of power networks by hypergraph cut algorithms. In *2014 IEEE International Conference on Smart Grid Communications* (2014).

[149] Yen, Jin Y. Finding the k shortest loopless paths in a network. *Management Science 17*, 11 (1971), 712–716.

[150] Zelke, Mariano. Weighted matching in the semi-streaming model. *Algorithmica* (2010), 1–20. 10.1007/s00453-010-9438-5.

[151] Zuckerman, David. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing 3*, 6 (2007), 103–128.