FINDING DENSE REGIONS OF RAPIDLY CHANGING GRAPHS

A Dissertation
Presented to
The Academic Faculty

By

Kasimir Georg Gabert

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2022

FINDING DENSE REGIONS OF RAPIDLY CHANGING GRAPHS

Thesis committee:

Dr. Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Srinivas Aluru
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. B. Aditya Prakash
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Ali Pınar
Department of Data Science and Cyber Analytics
Sandia National Laboratories

Dr. Srijan Kumar
School of Computational Science and Engineering
Georgia Institute of Technology

Date approved: April 21, 2022

For my family

ACKNOWLEDGMENTS

In no way would this work be possible with the tremendous support, encouragement, and direct help I have received over the years. I am extremely grateful to my advisor, Dr. Ümit V. Çatalyürek. Without his guidance, suggestions, ideas, and sincere support, this work would not have been possible. I am deeply thankful to Dr. Ali Pınar. I appreciate our numerous inspiring discussions that played a major role in my dissertation, and the helpful and important connections back to my Sandia work. I am thankful to the rest of my committee, Dr. Srinivas Aluru, Dr. Srijan Kumar, and Dr. B. Aditya Prakash, for improving my work through careful reading, helpful discussions, and suggestions.

This journey would not have started without the mentorship and encouragement of Tan Thai. I am deeply grateful for all you have done for my career and growth—much more than can be written in the words here. I thank Dylan Anderson, Dr. Stephen Todd Jones, Dr. Nick Pattengale, Dr. Laura Swiler, and Tom Tarman for the continual support, encouragement, and friendship at work during my studies. I am thankful to my managers Kim Denton-Hill, Cindy Veitch, Dr. Jennifer Troup, and Dr. Brandon Eames, who have all directly enabled my joint work and studies. I am thankful to Dr. Gayle Thayer for the tremendous help in allowing me to submit by conference deadlines.

I could not have gone through this without the support of my friends. I can't thank you all enough: Keegan Florence-Livoti, for many years of adventures and discussions; Srinivas Eswar for helping me get out and climb (among other adventures!); Dr. Abdurrahman Yaşar, for wonderful encouragement and discussions on all fronts; Dr. Nolen Scaife, for continuous support from when we both started; Bryan Kennedy, for a fantastic supply of interesting news updates (and discussions!); and Dr. Zhihao Li, for teaching me so much about how to capture light, a welcome break from studying. Matt, Mary, and Dennis: I cannot thank you enough for your support. I am thankful for all of the discussions, thoughts, and kindness from everyone in the TDA Lab, James Fox, Yusuf Ozkaya, M.

Fatih Balin, Kaan Sancak, and Ben Cobb, and the larger CSE program. I am thankful to Dr. David Bader and Dr. Oded Green for their early support, and all of my lab members.

My family has supported me the whole way, and I wouldn't be here without them. I am grateful to my parents, for their constant support; to my brothers, for keeping me in line; to my extended family, for being there for me; and to Jing, for everything.

Lastly, I am thankful for the parks, open spaces, and wild areas that allowed me to recharge and think more clearly.

This work was funded in part by the U.S. Department of Energy National Nuclear Security Administration's Office of Defense Nuclear Nonproliferation Research and Development (NA-22), the Doctoral Study Program at Sandia National Laboratories, and the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Many of today's massive and rapidly changing graphs contain internal structure—hierarchies of locally dense regions—and finding and tracking this structure is key to detecting emerging behavior, exposing internal activity, summarizing for downstream tasks, identifying important regions, and more. Existing techniques to track these regions fundamentally cannot handle the scale, rate of change, and temporal nature of today's graphs. We identify the crucial missing piece as the need to address the significant variability in graph change rates, algorithm runtimes, temporal behavior, and dense structures themselves.

We tackle tracking dense regions in three parts. First, we extend algorithms and theory around dense region computation. We computationally unify nuclei into computing hypergraph cores, providing significant improvements over hand-tuned nuclei algorithms and enabling higher-order nuclei. We develop new batch algorithms for maintaining core hierarchies. We then define new temporal dense regions, called core chains, that build on nuclei hierarchy maintenance and enable effective and powerful dense region tracking.

Second, we scale up on shared-memory systems. We provide a parallel input and output library that reduces start-up costs of all known graph systems. We provide the first parallel scalable core and hypergraph core maintenance algorithms, building on the connection between h-indices and cores. This addresses computation on rapidly changing graphs during bursty periods with large numbers of graph changes.

Third, we address scaling out to support massive graphs. We develop the first parallel h-index algorithm, the key kernel for tracking dense regions. We identify that system elasticity is imperative to handle large bursts of changes. We develop a dynamic and elastic graph system, using consistent hashing and sketches, and demonstrate competitive performance against static, inelastic graph systems while enabling new, dynamic applications.

By addressing variability directly—in algorithm and system design—we break through previous barriers and bring dense region tracking to massive, rapidly changing graphs.

CHAPTER 1
INTRODUCTION

Today's datasets are both massive and continuously growing. Numerous interactions can be recorded and stored by computer systems, such as computer network activity, social behavior, transportation events, biological and chemical processes, and many more. There is a large promise to this data: if we can understand what is happening inside of a system quickly, we can act to make positive change. In many cases, data includes internal relationships, and for those graph representations have proven powerful. Graphs are a powerful intermediate format: once data is put into a graph form, many domain agnostic graph algorithms can be effectively applied to reason about and understand the data.

Many of the large graphs today are *continuously changing* as natural processes and interactions progress in time. A graph representing websites and links between them, for example, changes as websites publish updates. Graphs representing social connections between people change as people change, and graphs representing computer traffic change when packets are sent. Between graphs, the rate of change can vary significantly. Some graphs may have only a handful of changes per year, for example academic coauthor graphs at a small institution, whereas others may have hundreds per microsecond, for example with network traffic graphs. Depending on the graph, the rate of change itself can be *highly variable* with bursty periods followed by relative calm.

It is easy to turn many real-world, massive data streams into graphs. The problem arises with *understanding* and *using* a resulting graph. As it rapidly changes, there is inherent error or staleness in parts of the graph that may impact algorithms' usefulness. When the graphs are massive, running an efficient, optimized algorithm from scratch may take minutes or even hours. If the graph is changing thousands of times per second during a bursty period, then even such an optimized static algorithm will not be able to run

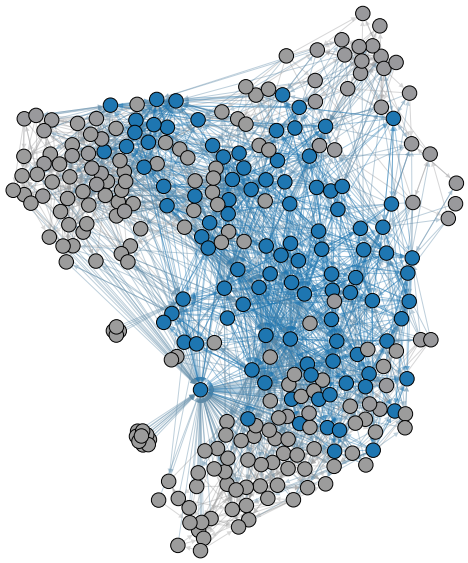


Figure 1.1: The dense core, shown in blue, of a neural network of C. elegans, a small worm. This dense region has been shown to contain important neurons for C. elegans' behavior. Many real-world graphs are globally sparse but contain locally dense regions.

quickly enough to return results before the graph has undergone significant, further change. This dissertation centers around taking a powerful, long standing technique to reason about static graphs—finding dense regions—and developing the theory, algorithms, and systems to enable this technique on massive, rapidly changing graphs, ultimately enabling deeper understanding and more effective use of these graphs.

Finding locally dense regions is a fundamental graph mining problem [155]. At a high level, dense regions expose parts of a graph that, by their nature, are robust against minor perturbation or error and due to their high internal connectivity may be more important than other regions. A surprising but important property of many real-world graphs is that while they are globally sparse they contain locally dense regions [260]. This means that finding dense regions can assist in graph summarization and identifying internal structure of graphs. In Figure 1.1, we show a dense region of the neural network of C. elegans [260], a small worm. Such dense regions have been shown to consist of neurons that are important in the behavior and central processing of the worm, whereas neurons in sparse regions make up peripheral and less critical parts of its brain [44, 253].

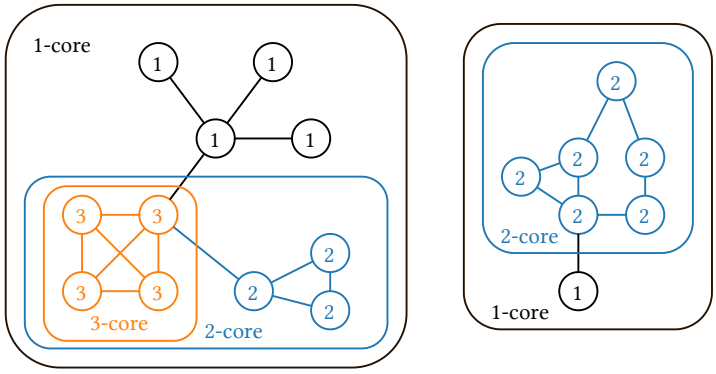
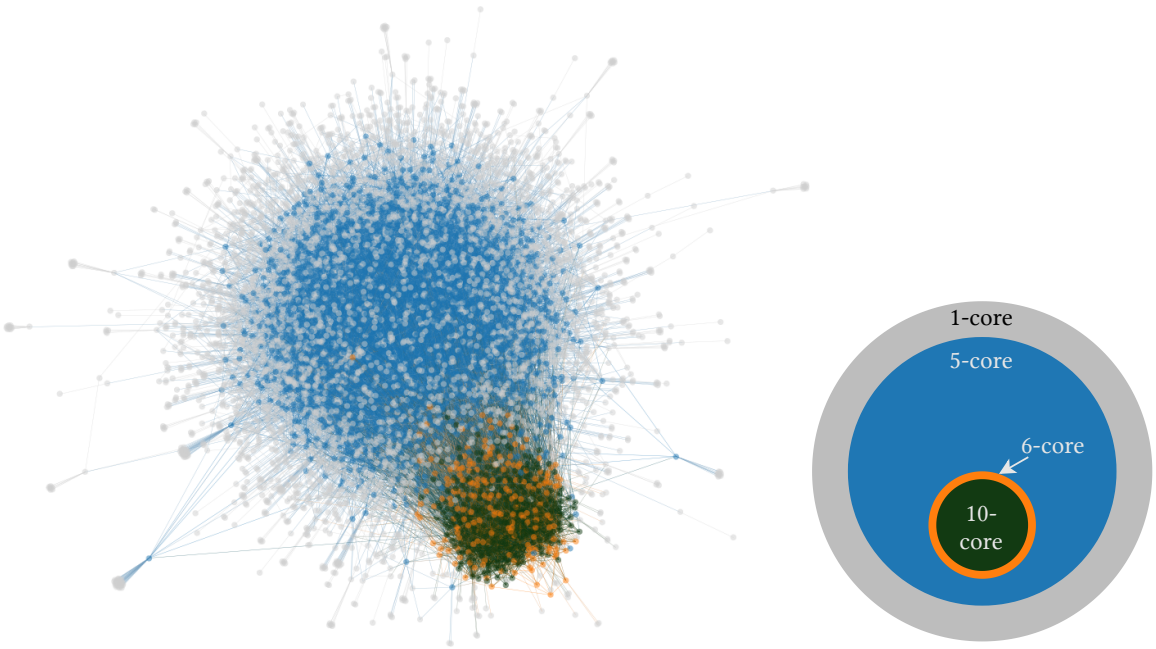


Figure 1.2: An example graph along with its k-cores.

Dense regions have been used for tasks including deriving news stories from blogs [13], finding communities in websites [148, 64], identifying link spam websites [103], uncovering motifs in DNA [88], performing graph visualization [10], optimizing computation [132], and many more [147, 9, 109, 251, 99, 85, 144].

Graphs that change over time are known as *dynamic graphs*, and they are referred to as *temporal graphs* if the history of the graph is saved and used. In this dissertation, we focus on the important problem of dense region *maintenance* on dynamic graphs and on finding *temporal* dense regions, by considering both historic and current information about the graph. We maintain decompositions and hierarchies through *dynamic graph algorithms*, find temporal dense regions with *temporal graph algorithms*, and we call *static graph algorithms* those which perform computation on an unchanging, immutable graph.

Finding the densest regions of a graph, those which are fully connected, is a classic NP-hard problem [138], and even finding less restrictive regions but those with explicit high density ratios are NP-hard problems and difficult to approximate [202, 227, 193, 39, 52, 220]. As a result, most dense region work has shifted towards *cores* [228, 183] or core-like regions [122, 50, 223, 264, 171, 28, 170, 272, 71, 271, 96]. A k-core is a maximal connected subgraph where every vertex has an degree at least k, and this degree constraint exists even when only the subgraph is present. In Figure 1.2 we show an example graph along with its k-cores. The graph is disconnected, and so has two 1-cores. Even vertices

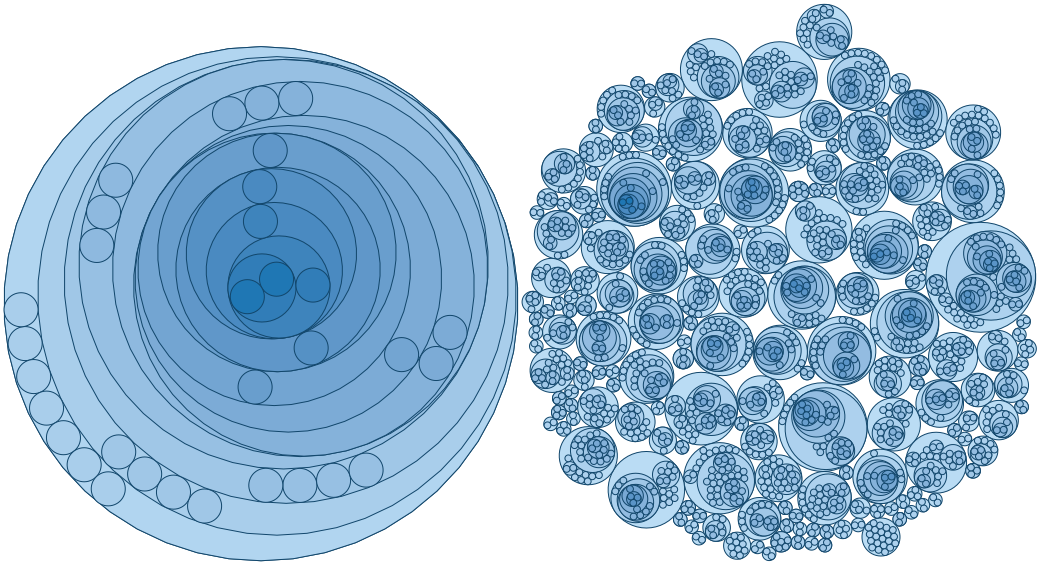


(a) The `Gnutella` graph. Colors correspond to four of its cores. (b) The core hierarchy.

Figure 1.3: The hierarchical and dense regions of a snapshot of `Gnutella`, a peer-to-peer network. Only 4 cores are shown for clarity. Cores are nested: each vertex with k neighbors also has $(k-1)$ neighbors, and so each k-core is also in a $(k-1)$-core. The sharp boundary between the 5-core and 6-core illuminate part of the peer-to-peer protocol with a corresponding gap between peers and "ultrapeers".

with a high degree may only be in a low core. The vertex has to both have a high degree and be in a *cohesive* region where enough neighbors also have high degrees. Cores can be computed in linear time efficiently [183] and, crucially, core decompositions can be *maintained* on medium sized, moderately rapidly changing graphs [274, 224]. Unfortunately, prior maintenance algorithms neither scale to rapidly changing graphs nor readily extend to more powerful dense regions.

In many cases, the dense regions are *hierarchical*: a very dense region may sit inside of a slightly less dense region, and so on [155]. In Figure 1.3 we show this holds for a peer-to-peer network, Gnutella [213]. Up until the 5-core the vertices are spread out uniformly, as shown in the visualization. The 6-core is a sharply smaller, more localized region. The following denser cores, up to the 10-core, are close to the 6-core. This hierarchical structure shows Gnutella's split between regular peers and "ultrapeers" [142].

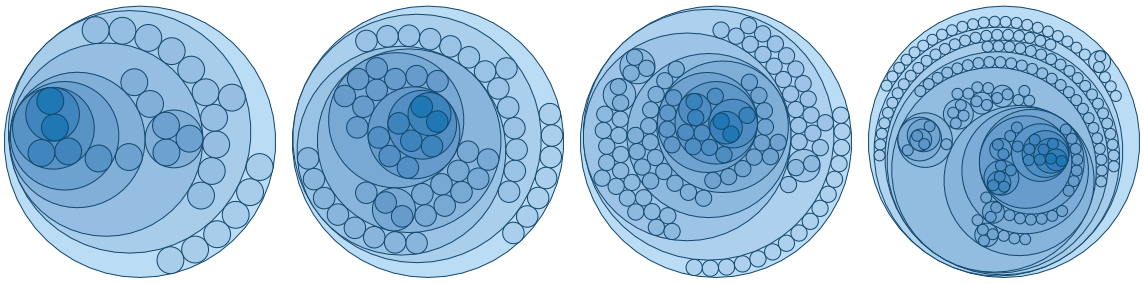


(a) Core Hierarchy (b) Nucleus Hierarchy

Figure 1.4: Extracting two hierarchies from the `web-Google` graph. Leaf nodes are removed and paths compressed for clarity. Each circle corresponds to a core or nucleus, and circles sitting inside correspond to nested cores or nuclei. Disjoint circles are disconnected cores or nuclei. Darker color indicates higher k values. In (a), a single funnel appears where there is one densest region. In (b), a rich hierarchy exists that, in many graphs, tends to reflect the nature of the graph. Cores are computable in linear time, but nuclei are denser, more cohesive and hence more useful.

Cores have received a significant amount of attention; however, on many graphs they fail to capture useful hierarchies [223]. In many cases they return a single "funnel", with one densest core and overlapping larger and larger cores out of it. Nuclei [223] generalize cores by replacing the *degree* requirement with a *clique-degree* requirement, where a k-clique is the complete graph on k vertices. As we discuss later, nuclei are parameterized by two integers, and for the examples here we choose the parameters $(3, 4)$. With nuclei, dense regions have stronger and tighter connections. In Figure 1.4, we show the `web-Google` [107] graph, the core hierarchy, and the nucleus hierarchy. When moving from cores to nuclei, the regions become denser and more cohesive. The resulting hierarchies become richer and ultimately provide more insight into the graph.

Extracting dense regions have proven highly useful on static graphs. However, only limited forms of dense region maintenance—specifically for cores [225, 274] and trusses [122,



(a) 20% (b) 40% (c) 60% (d) 80%

Figure 1.5: The nucleus hierarchy of `facebook` at various points in time, given in percentages of the edge stream. Dense regions can show temporal changes motivating the need to compute dense regions at various points in time on temporal graphs.

273], an extension of cores—have been developed, and those approaches do not scale [121]. There is a significant need for scalable and effective techniques to maintain cores and other dense regions on today's massive, rapidly changing graphs.

Furthermore, the hierarchies of graphs and the dense regions change over time, and tracking these changes has the potential to yield significant insight. In Figure 1.5 we show a dense hierarchy of the `facebook` temporal graph. There have been several approaches that study temporal cores, however prior approaches produce brittle core structures that do not fully consider the temporal nature [18, 165, 123, 97, 206, 208, 167, 166, 173, 216]. Instead, we need new definitions that capture the temporal nature of cores themselves.

Unfortunately, simply engineering prior maintenance algorithms to handle larger scales does not work. There are several major problems: first, core maintenance algorithms are complex and porting them to other core-like dense regions is both error prone and prohibitively expensive, and prior temporal approaches fail to capture important structure; second, we prove any nucleus maintenance algorithm necessarily has worst-case runtimes that match a static computation, so with massive graphs runtime variability must be addressed throughout algorithm design; third, prior maintenance algorithms are either not parallel or not scalable. In this dissertation we introduce new algorithms and systems that address these problems and provide effective dense subgraph extraction on massive, rapidly changing graphs.

1.1 Thesis Statement and Research Overview

There are two recurring themes addressed throughout this dissertation.

1. There is significant *runtime variability* in dynamic graph algorithms, both due to variability in the rate of change of the graph and theoretically necessary variability in computation, even when the output does not change at all.

2. As graphs change, there is variability in the *dense regions themselves*. This variability can be harnessed to provide further, temporal insight about graphs.

We argue that *by handling variability explicitly throughout both algorithm and system design, hierarchies of dense regions can be maintained on massive, rapidly changing graphs.*

We address the gap in three parts: extending and unifying notions of density to enable effective and more powerful computation; scaling up shared-memory algorithms and systems; and scaling out to distributed systems to support massive, rapidly changing graphs.

Part 1: Extending and Unifying Dense Regions. First, we extend and unify notions of dense regions of graphs. Without this work, the power and ability to use dense regions is significantly reduced.

In Chapter 3 [90], we address the problem of dynamically maintaining nuclei. Computing nuclei involves counting and enumerating (small) cliques, where the clique size is a parameter. In many cases, larger cliques result in higher computational costs yet, at the same time, provide richer and more useful hierarchies. As such, computing nuclei can be a major undertaking: the cost can be prohibitively high if the graph contains large cliques and undergoes continuous change, with a high cost incurred on every update. These costs result in a pressing need for maintenance algorithms.

Our first result is somewhat negative: we prove that in the worst case maintaining nuclei, even on a single edge update with no changing output, cannot asymptotically beat

re-computing from scratch. This motivates studying the *empirical runtime* of maintenance algorithms and shows that long tails and high variability need to be addressed.

Prior to our work, there were no known approaches for maintaining general nuclei. Specialized algorithms have been developed for two common nuclei, $(1,2)$ and $(2,3)$. However, each specialization is involved with no clear path to higher-order nuclei.

We developed a simplified way of thinking about nuclei: instead of by their original definition, we view all nuclei as $(1,2)$-nuclei, or cores, in a special hypergraph. This result immediately simplifies both nuclei descriptions and computational approaches. To maintain nuclei, we can now use hypergraph core maintenance algorithms. Using order-based methods, we developed a new hypergraph core maintenance algorithm that is exact and orders of magnitude faster than prior approximate-only methods, resulting in 7 orders of magnitude improvement over re-computing nuclei from scratch. Furthermore, not only do we maintain higher-order, arbitrary nuclei, but our implementations work faster than specialized truss algorithms—over 93× faster. We expect that, through our framework, as faster core maintenance algorithms are developed they will be amenable for computing arbitrary nuclei, bringing performance improvements with them.

In Chapter 4, we focus on a critical but largely overlooked issue: all prior maintenance algorithms have maintained the *density levels for vertices*, but have ignored returning the *set of vertices* in a dense region. This oversight means that prior maintenance algorithms cannot be used for most downstream applications, as they require the actual vertices, and this severely limits the applicability and usefulness of finding dense regions in graphs.

To address this, we use a query-efficient shell tree index from community search that can maintain connectivity and hence the sets of vertices. Our approach not only applies to graph cores, but also to hypergraph cores, and so we are able to return the dense regions for nuclei. Differing from community search results, we use an intermediate data structure. We build and maintain a directed acyclic graph where each node represents a connected region of the graph with *identical density values*. This directed acyclic graph is then used

to create the shell tree index, allowing for batches with concurrent insertions and deletions and work savings over a direct application of the community search results.

In Chapter 5, we move beyond only looking at maintenance algorithms and focus on *temporal dense regions*. These are regions that take advantage of the temporal nature of the graph: instead of trying to simply maintain the dense region, computing updates as quickly as possible, the goal is to understand how a dense region, and ultimately the graph, evolves over time. Can we follow a dense region from cradle to grave?

While there has been a recent large surge in interest in solving this problem [18, 165, 123, 97, 206, 208, 167, 166, 173, 216], we demonstrate that all of the known prior approaches suffer from a particularly deep-rooted issue. They start with a static graph definition of a dense region, for example a core is defined as a maximal subgraph with a minimum degree at least k, and then look for *time intervals* for which any such subgraph exists. Some approaches look for rapid bursts that create the subgraph, others for periodic behavior, diversified subgraphs and vertices, and yet others for longest living subgraphs. Unfortunately, all of these approaches cannot handle subgraphs that themselves change over time. Critically, *dense regions themselves* can outlive vertices and edges.

To address this problem, we define *core chains* as a path through different density hierarchies over time. We treat such a path itself as a dense region and the goal is to compute useful paths. We develop two core chains that work with core and nuclei hierarchies: one which focuses on a seed set of vertices and the other which focuses on the majority of vertex movements from timestep to timestep. Unlike prior approaches, we show that core chains are able to identify dense regions that change over time. In a co-author graph, we show that nuclei-based core chains are able to identify research groups and can even distinguish between two closely related research groups. In a dataset collected from observing ants, we show that core chains are able to successfully identify ant behavior and can even assist in identifying what roles ants play in the colony. In both cases, prior temporal core approaches cannot extract any useful regions.

Part 2: Scaling Up. As datasets become large, single threaded approaches are no longer sufficient. Second, we address scaling up in shared-memory environments.

In Chapter 6 [89], we address the first problem that many graph systems, including our dense region work, face when scaling up: input and output. Prior to our work, the few available input and output libraries were single threaded and generally slow. However, modern architectures support parallel input and output, and furthermore slow data parsing tends to become a serious bottleneck. For graphs with tens of billions of vertices, input can take upwards of an hour, whereas kernel runtimes then complete in seconds.

We developed an open-source library, PIGO, which reads graph and tensor data, both in ASCII and in binary formats, in parallel. When handling ASCII files, it makes two passes: in the first pass, offsets are computed in parallel and memory is allocated; in the second pass, the actual values are copied, again in parallel. We modified the most popular graph libraries to use PIGO and we achieve end-to-end improvements of up to 38× over running without PIGO. We have continued to support PIGO and it has been used in a variety of both small and large internal and external projects.

In Chapter 7 [91], we then focus on shared-memory parallel maintenance of cores. We address both cores and hypergraph cores, enabling nuclei maintenance in parallel. There have been several prior parallel core maintenance approaches, and the general strategy has been to identify *independent parts of an update*, and then process those parts concurrently. This is a problematic approach because there can be significant connections between many parts. Prior work has shown that these approaches by and large do not improve runtime as additional threads are added—speedups of only around 1.5× with 64 cores.

We take an entirely different approach, and provide two scalable and effective algorithms to maintain cores. Both algorithms build on the connection between h-indices and cores. The first algorithm identifies a large region of the graph and pessimistically increments all of the values inside of it. This region may be chosen due to vertices sharing the same initial density value, or being in the same connected component, among others.

Critically the region must be fast to calculate and increment. Then, in parallel, all vertices iteratively compute h-index values with their neighbors' values until convergence. This approach ends up both outperforming prior approaches by over 4× and scales well, up to 13× speedup with 16 threads, but comes with significant costs when batches are small.

Our next algorithm takes a separate approach and optimistically assumes that nothing in the graph will change. Instead, it attaches an identifier to each change in the batch and then passes that identifier around the graph, while concurrently iteratively computing h-indices. In the worst case, this may double the convergence time, but it may also result in significantly less work when changes can be quickly restricted to a small, local region. For small batches, this approach similarly scales well, up to 11× with 16 threads.

In both cases, when addressing hypergraphs there are a variety of significant additional concerns and considerations that need to be handled. Overall, our two algorithms are the first scalable core maintenance algorithms and enable dense region maintenance on graphs that fit into shared-memory systems.

Part 3: Scaling Out. Third, we focus on scaling out on distributed memory systems. When graphs are rapidly changing and sufficiently large, or have numerous clients querying them, they need to be stored on and processed with distributed systems.

Chapter 8 focuses on computing h-indices in a distributed manner, which is a critical kernel for computing cores and hence nuclei. Prior to this work, the only known approaches were sequential and based on sorting. We developed a new approach to computing h-indices that pivots on the input and recurses to one side, similar to fast selection algorithms.

We implemented our algorithm and tested it sequentially, on shared-memory systems, and on distributed-memory systems. With large inputs we are faster than prior algorithms; running sequentially, we are even over 1.5× faster, and we scale well from there. In distributed-memory environments we scale to massive lists, returning the h-index from a list with 3 trillion integers in under 10 using 6784 cores.

In Chapter 9 [93], we focus on developing the first distributed, dynamic graph system built for rapidly and variably changing graphs. Previous approaches have largely ignored what we identify as a critical problem, that of elasticity. Many graphs with high rates of change do not change uniformly over time; instead, they change in both bursty periods and calm periods. Prior approaches were unable to scale at runtime, and so they either over-provision during calm periods or are unable to keep up and sustain activity during bursty periods. Many prior systems focused heavily on partitioning to reduce the kernel runtimes for breadth-first search or PageRank, which makes elasticity appear only as a distant hope.

We take a different approach. We use two layers of consistent hashing coupled with a degree sketch to perform elastic partitioning. We then use a shared-nothing architecture to store and process the graph. We built our system, named ElGA, using typical cloud assumptions, and designed it to perform batch processing with dynamic edge changes.

Overall, ElGA ends up a serious competitor for distributed graph systems, independent of dynamic or elastic needs. Even excluding our elasticity and dynamic graph support, we have faster per-iteration runtimes than state-of-the-art static systems (that use high-performance computing technologies, such as highly tuned MPI libraries): we are over 2.4× faster. Furthermore, we do not incur any separate partitioning costs. We can elastically scale up and down, and support edge rate changes of around one million updates per processing core. Running ElGA on only a single shared-memory system, we maintain connectivity 1.2× faster than shared-memory specific dynamic graph systems. Through ElGA we show that our design is capable of scaling to hundreds of billions of edges, supporting thousands of concurrent clients, enabling infrastructure scaling, both up and down, as needed, and effectively computing graph analytics including dense regions.

In summary, we provide a comprehensive set of algorithms, tools, and techniques that enable *extracting dense regions* from rapidly changing graphs. Our approaches algorithmically unify core-like dense regions, provide effective temporal dense region targets, scale up to take advantage of shared-memory, and scale out to handle massive dynamic graphs.

CHAPTER 2

PRELIMINARIES AND NOTATIONS

In this chapter we present preliminary material and notations that are used throughout the dissertation. We first formally describe the graphs that we consider and their properties, dynamic graphs, and finally dynamic graph algorithms. We then describe different definitions of density, various core and core-like regions, and the computational strategies for computing cores. Finally, we describe large-scale distributed graph systems and focus on their ability to support variability.

2.1 Graphs and Graph Extensions

A graph is a powerful abstraction that emphasizes the *relationships* within data, as opposed to the explicit values.

Definition 2.1. *Let V be a set and $E = \{e = \{u, v\}\}$, where $u, v \in V$. Then $G = (V, E)$ is a graph.*

The set V consists of *vertices* and E consists of *edges*. These graphs are sometimes referred to as a *simple graphs*. While it is easy to put data into this form care needs to be taken. For example, a computer network data can place IP addresses as vertices and packets between IP addresses as edges, however just as easily edges could be sufficiently large packets or long-lived network streams. In many cases the construction of the graph—choosing what to use as vertices and what to use as edges—can play a significant role in what a graph algorithm will output.

While graphs prove to be a useful abstraction, in many cases there is more data available and some algorithms have been developed to take advantage of them. Following we list several extensions which our dynamic graph systems support.

Directed graph *Orient* each edge $e \in E$, such that $e = (u, v)$, where u is the *source vertex* and v is the *destination vertex*.

Multigraph Allow multiple edges two exist, that is change E from a set to a multiset.

Weighted graph Each edge is augmented with a weight, which is typically either a real-valued number or an integer.

Temporal graph Each edge is augmented with two timestamps, one of which corresponds to the edge creation time and the other an edge deletion time.

A *hypergraph* is a generalization of graphs where the edges are not restricted to only consisting of two vertices. These are important to address as they are both natural in many datasets—those where relationships exist between *multiple* vertices—and, as we show later, can be an important tool for dense graph construction.

Definition 2.2. *Let V be a set and $E = \{e \subseteq V\}$. Then $H = (V, E)$ is a hypergraph.*

Hypergraphs are highly related to, but slightly different, from *bipartite graphs*.

Definition 2.3. *Let V_1, V_2 be two sets and $E = \{e = \{u, v\}\}$, where $u \in V_1$ and $v \in V_2$. Then $B = (V_1, V_2, E)$ is a bipartite graph.*

If the hypergraph is naturally extended to a multihypergraph (where E is a multiset that can have empty subset elements), then it is easy to see they are equivalent to bipartite graphs.

Our algorithms for finding dense regions and their hierarchies are designed for simple graphs and hypergraphs only, as they are the most widespread form of graphs.

2.2 Graph Properties

There are many properties of graphs and we clarify the ones used throughout this dissertation here. Let $G = (V, E)$ be a graph and $u \in V$.

Degree The degree of u, $\deg(u)$, is the number of edges u participates in, that is $\deg(u) = |\{e : u \in e, e \in E\}|$.

Neighbors The neighbors of a vertex are those which are connected via edges to a vertex. The neighbors are denoted $\Gamma[u]$

Subgraph A graph contained within another graph, that is $S = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$.

Induced Subgraph A subgraph formed by taking all well-defined edges given a vertex set. Let $V' \subseteq V$. Then, $G[V'] = (V', \{e = \{u, v\} : e \in E, u \in V', v \in V'\})$ is an induced subgraph.

These properties similarly hold in hypergraphs.

2.3 Dynamic Graphs

Throughout this dissertation we consider undirected, simple graphs that are changing over time, known as dynamic graphs. Let V be an infinite set of vertices. We view these graphs as an infinite turnstile stream $\mathcal{S}$ of edge changes, where each change $\langle e, c \rangle$ consists of a pair of vertices that change, $e = \{u, v\}$ with $u \neq v \in V$, and a change type, $c \in \{+, -\}$, where + indicates an edge addition while − indicates removal. For notational convenience we denote the subsequence in the range i to j by

$$\mathcal{S}_i^j = \left(\langle e_i, c_i \rangle, \langle e_{i+1}, c_{i+1} \rangle, \ldots, \langle e_j, c_j \rangle\right) \subset \mathcal{S}.$$

We call time the position in the stream, so at time t the graph $G^{(t)} = (V^{(t)}, E^{(t)})$ can be derived by applying $\mathcal{S}_0^t$ in order starting from the empty graph $G^{(0)} = (\emptyset, \emptyset)$. $V^{(t)}$ is the finite vertex set containing vertices in edges seen by time t. Throughout we assume that for all $i \geq 0$, $G^{(i)}$ is an undirected simple graph.

Note the similarity between dynamic graphs and *temporal graphs*, defined above. Both change over time. Hoever, in a temporal graph, the history of all edge insertions and removals—along with the change time—is preserved. This can provide much more information to an algorithm, however it is a challenge to make effective use of this information [118]. Most temporal graph systems are not built to return real-time results, and so are not suitable for dynamic algorithms or processing dynamic graphs.

From a system perspective, temporal graphs and dynamic graphs have different goals: in a dynamic graph, *timeliness* or the latency of a graph update is most important. In temporal graphs, retaining history and supporting queries that may look back through regions of time is most important. In this work, we target both *dynamic graphs* and *temporal graphs*, through the lens of dynamic graph algorithms. In particular, we show that it is possible to derive interesting and powerful temporal results by applying dynamic graph algorithms for each timestep, which relies on the timeliness of dynamic graph algorithms for efficiency reasons.

Another closely related concept is that of *streaming graphs*. These are also graphs that change over time. However, a *streaming graph algorithm* differs from a *dynamic graph algorithm* in that it has significant memory and runtime constraints. Streaming graph algorithms operate *only* in the turnstile stream of the dynamic graph. This means that they need to use $o(|E|)$ memory and are typically constrained to a small number of passes over the streaming, meaning a runtime of $\tilde{O}(|E|)$.

2.4 Dynamic Graph Algorithms

An algorithm that operates on a dynamic graph stream is a *dynamic graph algorithm*. An algorithm operating on a stream with only insertions is *incremental* and one operating only on deletions is *decremental*. Let $\mathcal{A}$ be a graph algorithm which, given an input graph G, produces an output $\mathcal{A}(G)$. We call $\mathcal{A}_\Delta$ a dynamic graph algorithm if it takes as input output from a prior time, say time i with $i < j$, along with all changes in the stream up to

time j and produces the correct output for the graph at time j. That is, $\mathcal{A}_\Delta$ takes as input $(G^{(i)}, \mathcal{A}(G^{(i)}), \mathcal{S}_i^j)$ and produces as output $(G^{(j)}, \mathcal{A}(G^{(j)}))$. This algorithm can then be repeatedly called as the stream progresses. We call $\mathcal{S}_i^j$ a batch. The *latency*, or runtime, of $\mathcal{A}_\Delta$ for $\mathcal{S}_i^j$ is $T_{\mathcal{A}_\Delta}$, and the *throughput* is $|\mathcal{S}_i^j|/T_{\mathcal{A}_\Delta}$.

A maintenance algorithm is used when latency is more important than throughput. Any reduction in latency is pursued, which commonly reduces throughput. The goal is to develop an algorithm with as small a latency as possible yet a corresponding throughput capable of sustaining the natural rate of change in the graph.

The following terms are used when referring to dynamic graph algorithms.

Query Latency Query latency is the round-trip-time for a client to request a computation result for the latest batch.

Query Staleness Query staleness is the time difference between the return time of the query and any update in the graph and algorithm output. Any positive value means that a newer result could have been delivered instead, if the system were faster.

Graph Throughput Graph throughput is the number of edge changes per second that are modified in the graph.

Batch Latency Batch latency is the total computation time for all processors to complete processing and store results for the given batch.

It is important for query latency and staleness to be low. The graph throughput needs to be high for rapidly changing graphs. The batch latency is the main computational target evaluated in dynamic graph algorithms, however it is not directly relevant to the end user. However, a low batch latency results in a low query staleness.

2.5 Dense Regions of Graphs

2.5.1 Definitions of Density

Throughout this dissertation, when we refer to density we are considering *edge density* [21].

Definition 2.4. *Let $G = (V, E)$ be a graph and $A \subseteq V$ be a set of vertices. Let $S = G[A] = (A, E')$ be the induced subgraph of A in G. Then, the edge density of S is given by*

$$\frac{|E'|}{\binom{A}{2}} = \frac{2\,|E'|}{|A|\,(|A|-1)}.$$

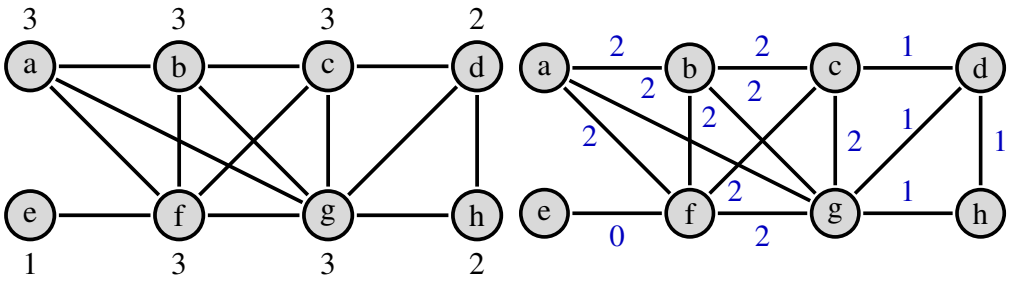
Defined in this way, density is one with a k-clique, where all possible edges exist. As the subgraph loses edges, density decreases. There are other forms of density, for example average degree density defined as $|E'|/|A|$, where E' and A are defined as in Definition 2.4. Whereas finding the edge densest subgraph is NP-hard, finding the average degree densest subgraph is solvable in polynomial time [104]. An interesting problem is finding the computational boundary between the average degree density and edge density problems [254].

We do not solve either problem directly, and instead focus on linearly solvable objectives, namely cores and core-like regions, described next. These have become the standard technique for finding dense regions on large graphs, and have proven useful for many applications [155, 180].

2.5.2 Cores, Trusses, and Nuclei

A k-core of a graph [228, 183] is a maximal connected subgraph where each vertex has an induced degree of at least k. The *coreness* of a vertex is the largest k such that the vertex is in a k-core. Figure 2.1a shows the coreness values on an example graph.

Note that a *decomposition* will compute only the coreness values. A full hierarchy is considered a separate operation. For the most part, k-core, k-truss, and other algorithms



(a) Coreness (b) Trussness

Figure 2.1: A sample graph with its coreness (a) and trussness (b) values shown. Consider any induced subgraph with $\geq k$ coreness (resp. trussness): the degree (triangles per edge) will be at least k for all vertices (edges).

only focus on the decomposition and leave the full hierarchy computation, which would return the actual dense regions, as a separate operation. We address this gap later in Chapter 4. It is too expensive to compute the hierarchy in a naive manner, and static hierarchy computations are built using disjoint-set approaches [221].

A k-truss community of a graph is also used to find important dense subgraphs [122]. A k-truss community is a maximal subgraph where each edge is contained in at least $(k-2)$ triangles, and each edge is connected to all other edges through a path of triangle inclusions. In this chapter, we use a slightly different definition to achieve a simple and unified generalization, and define k-truss community as a maximal subgraph where each edge is contained in at least k triangles, and keep the same connectedness requirement. A k-truss, but not a k-truss community, is an earlier, similar concept without connectivity constraints [50]. The community trussness of an edge is the largest k such that an edge is part of a k-truss community. Figure 2.1b shows the trussness values on an example graph. Trussness has also been computed using peeling algorithms [257, 122, 231, 50] in $O(|E|^{3/2})$.

Coreness and community trussness may seem only related at an algorithmic level. Sarıyüce et al. showed that there is a deeper connection: by viewing a k-core as a cohesive unit where 1-cliques are contained in 2-cliques, and k-truss communities as cohesive units where 2-cliques are contained in 3-cliques, the concept was generalized to nuclei [223].

Numerous other targets, similar to cores, have been proposed [180]. [71, 275] develop

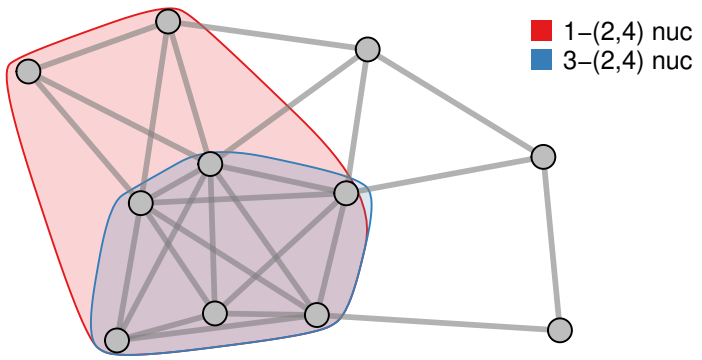


Figure 2.2: A $(2, 4)$ nucleus decomposition. Every edge in the 3-(2,4) nucleus has three distinct 4-cliques it is part of in the nucleus-induced subgraph.

weighted extensions to cores, [170] uses core concepts to reinforce connections within networks, [96] proposes notions of cores for multilayer networks, and [271] ensures vertices in core-like regions are also relatively cohesive given their neighbors. In cases where the cores are used for downstream algorithms, returning the actual (connected) vertices is identified as crucial and algorithms are built to support such queries [171].

To understand graph nuclei, let r, s with $1 < r < s$ be two integers. Recall an l-clique is a fully connected graph with l vertices. Each s-clique internally contains $\binom{s}{r}$ r-cliques. A subgraph $N \subseteq G$ is a k-(r, s) nucleus of a graph if the following hold:

- it is a union of s-cliques,
- it is maximal,
- each r-clique in N is contained in at least k distinct s-cliques in N, known as the S-degree or support, and
- each pair of r-cliques can be connected by traversing through s-clique inclusions, known as S-connectivity.

The nucleus decomposition value is defined for r-cliques. It is the largest k such that the r-clique is in a k-(r, s) nucleus but not in any $(k + 1)$-(r, s) nucleus. We denote this value $\kappa[x]$ for r-clique x. Note that if an r-clique is in a k-(r, s) nucleus, then it is also in

Algorithm 2.1: The peeling algorithm for computing coreness values. The linear time comes from a bucketing data structure to iterate through vertices, sorted by $d[v]$, while modifying $d[v]$.

Input: graph $G = (V, E)$
Output: τ

```
1  ∀v ∈ V, d[v] ← deg(v)
2  k ← 0
3  for v sorted by d[v] do
4  |  if d[v] > k then
5  |  |  k ← d[v]
6  |  for n ∈ Γ[v] do
7  |  |  d[n] ← d[n] − 1
8  |  τ[v] ← k
9  |  delete d[v]
10 return τ
```

a $k' \leq k$ nucleus. Additionally, the connected components of nuclei at different k levels form a laminar family. This property means that given κ and connectivity information, a hierarchy can be formed that contains every nucleus and its r-clique membership [221, 80]. Therefore, we are interested in computing κ for each r-clique. The support of an r-clique, denoted sup, is the number of distinct s-cliques that it is contained in. An example showing nucleus decomposition values is shown in Figure 2.2.

2.5.3 Static Computation

The most straightforward way of computing cores is through "peeling." [183] In this approach, vertices are iteratively removed—typically by keeping track of whether they are active or not, not by modifying the graph—if their degree is less than k. Any vertex that is removed has its k-core value assigned as k. When all vertices are removed, the process stops. If vertices are organized in buckets based on their degree, this process can be done in $O(|V|+|E|)$ [183]. Variants of this approach are used for processing in parallel [56, 135, 58]. Parallelization in these approaches largely occurs by taking advantage of parallelism within levels. A core peeling algorithm is shown in Algorithm 2.1.

A separate approach follows the connection between h-indices[116] and k-cores [174].

Algorithm 2.2: An h-index algorithm to compute coreness values.

Input: graph $G = (V, E)$
Output: τ

$\forall v \in V, \tau[v] \leftarrow \deg(v)$
$t \leftarrow 0$
repeat
  for $v \in V$ **do**
    $L \leftarrow \left\langle \tau^{t-1}[w] : w \in \Gamma[v] \right\rangle$
    $\tau^t[v] \leftarrow$ H-INDEX(L)
  $t \leftarrow t + 1$
until τ *no longer changes, converging to core values*
return τ

This strategy was first developed as a distributed algorithm [196] and later shown to be close to state-of-the-art in a shared-memory setting [222]. The idea of this approach is to iteratively update a *local value* associated with each vertex. The local value on vertices is initialized high, and in each iteration the h-index of each vertices' neighbors' local values is computed. The process terminates when changes are made. The advantage of this approach is that each vertex can operate independently, asynchronously. This algorithm is shown in Algorithm 2.2.

2.5.4 Hypergraph Cores

In hypergraphs a core is defined exactly the same way: it is a maximal connected subgraph with minimum degree at least k. Similarly, a k-core value for a vertex in a hypergraph is the largest value k such that a vertex is in a k-core but not in any $(k + 1)$-core. An example hypergraph and its k-core decomposition is shown in Figure 2.3.

Cores have proven most useful with co-occurrence hypergraphs in social network analysis [243], however work on them is only now beginning [233]. Cores have proven crucial for specialized, derived hypergraphs in erasure codes [175], bloom filters [192, 45, 106], satisfiability problems [38, 194], and hashing [200].

(α, β)-cores [62, 171] are developed for bipartite graphs, and essentially allow for two,

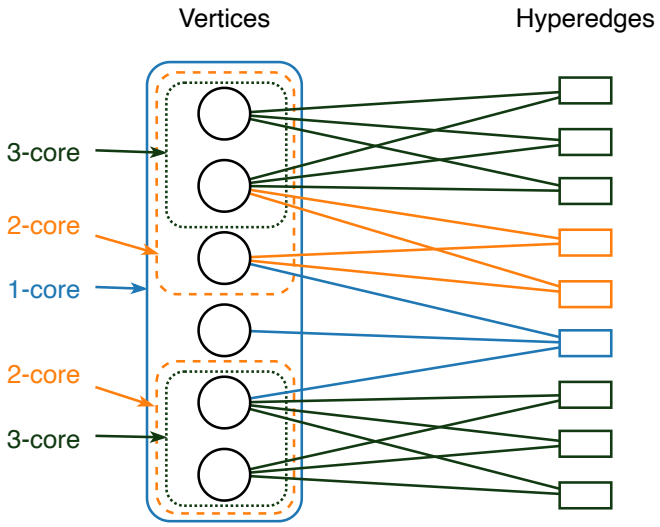


Figure 2.3: An example hypergraph and its cores.

separate k values for α-cores from V_1 and β-cores from V_2. This is separate from hypergraph cores as in hypergraph cores an edge is either completely included in the core, or completely excluded. In (α, β)-cores, a vertex may have only some of its (bipartite graph) edges included.

In hypergraphs, the standard peeling approach works as well, including in parallel [233]. Handling k-cores in dynamic hypergraphs has seen recent attention [243]. Here, the authors identified the need and demonstrated the importance of solving hypergraph k-cores. Due to the runtime of prior graph approaches, they present only an approximate sequential solution, again based on peeling.

2.5.5 Maintenance Approaches

There are two main approaches for *maintaining* cores on dynamic graphs. The first is known as the traversal algorithm [225, 164]. Given a new edge, it performs a depth-first graph traversal from the endpoint with the lowest coreness. The traversal remains within a *subcore*, which is a connected region with the same coreness value. If it comes across a vertex unable to increase, it stops searching that path. The second is known as the order algorithm [274]. A valid order, or decomposition order, is an ordering of vertices such that

if they are deleted, the coreness On an edge insertion, this algorithm inductively assumes the vertices are in a valid "peeling" order. The order based approach was extended, in the *truss* setting, to handle batches [273]. This approach is not parallel, but takes advantage of overlapping work inside of batches and uses that to improve performance.

Parallel approaches have relied on identifying a set of vertices that can be independently peeled [131, 6, 121, 14]. Unfortunately, on real-world graphs those opportunities are limited, and these approaches suffer major scalability problems as the number of cores grow. Concurrent to our work, [98] considers h-index based methods for dynamic core maintenance. Unfortunately, this work is not parallel and the approach is not competitive against state-of-the-art sequential methods [274, 273].

Bai et al. [19] and Zhang and Yu [273] propose batch algorithms that reduce work as multiple edges are processed simultaneously.

2.6 Large-Scale Dynamic Graph Systems

In this section we provide an overview of both distributed graph systems with a focus on dynamic systems, along with the closely related dataflow systems. We focus on the *elasticity* of the systems, as this determines whether they can effectively scale to handle highly variable workloads at runtime.

There are two main categories of graph systems: performance focused systems and database systems. The goals and inputs of each type of system vary. In performance focused systems, graphs will typically be read from the disk in a format that only includes their structure. The computation will run as a batch operation and save the results back to disk, before shutting down the runtime. In a graph database system, the graph will typically persist in memory and handle changes. In these systems, the queries are typically lightweight and easy to process, and the challenge is in efficiently and resiliently storing the graph along with the associated attributes. There are a significant number of highly impactful performance focused static graph systems and database graph systems [217, 184].

2.6.1 Dynamic and Partially Dynamic Graph Systems

There are numerous distributed graph systems that have been proposed and developed and many have some notion of a changing, or dynamic, underlying graph. A thorough review of prior systems is given in [26]. Here, we provide a brief overview to highlight the gap in dynamic, elastic systems. We call a system *partially dynamic* if re-running from a previous output is possible, and *fully dynamic* if it supports low-latency queries and is built to handle a rapidly changing graph.

ChronoGraph [74] is a fully dynamic system built in NodeJS, but not elastic. An evaluation framework for any stream-based graph system is provided in [73]. An elastic system for data stream processing is given in [101], but it does not support dynamic graphs. Similar to other cloud systems, it uses consistent hashing, which we rely on as well. However, it is a generic stream processing framework and not designed to handle graphs, in particular high-degree vertices on scale-free networks. EdgeScaler [204] adds a elastic partitioning scheme to static graph systems. However, it does not extend to incremental or dynamic graphs.

GraM [265] has a follows a shared-nothing architecture. It performs message passing, allowing it to scale up on a single machine as well as out to a cluster. However, it is tuned to use low-level features of NICs, such as RDMA, which may not be available in general cloud environments. Furthermore, it does not have support for elasticity or dynamic graphs.

Kineograph [46] is a dynamic graph system that provides a simple programming model to operate on snapshots of the graph. It has support for replication, and outlined the need for elasticity along with a rough approach. In [195], a dynamic graph system was built on top of CouchDB. This work focused on understanding replication strategies and factors to minimize traversals and optimize load imbalance. Concerto [154] uses elastic key-value stores internally, but does not explicitly support scaling. It stores graph updates in custom data structures. While it inherits some fault tolerance support from underlying key-value stores, it is not elastic and the fault tolerance is not carried upwards into the graph process-

ing itself. GraphTau [127] is a dynamic graph system built on top of Spark. It has support for recovering from failures by periodic checkpointing. ZipG [140] stores graph updates in a compressed format. This compressed format is capable enough to handle a large number of queries, and so the number of machines required is low. UNICORN [244] is a dynamic graph system built on top of IBM InfoSphere Streams. CellIQ [126] is a domain-specific dynamic graph system optimized for analytics on cellular phone networks and is built on top of GraphX. KickStarter [256] takes a different approach to dynamic graph processing by inexpensively maintaining an approximate computation. Then, if the accurate computation is required, it can try to skip the intermediate maintenance steps and provide a full result sooner. Delta-BiGJoin [11] is an approach implemented in Timely Dataflow [198] that is optimized for efficient subgraph queries. Sprouter [2] is a dynamic system built on top of Spark that uses both Spark's streaming and batch processing APIs. Sallinen et al. [218] propose a model where edge changes are timestamped and answers are retained for each observed timestamp. None of these address elasticity.

GRAPE [79] is a distributed graph system that can be extended to support incremental computation and operates in a variant of a local computation model. However, it requires the graph to be accessible on a single master machine, breaking typical memory requirements of distributed graph systems. Further work [76, 78] has improved the system to use consistent hashing to allow for dynamic scaling of machines, similar to our overall approach. However, this does not take a shared-nothing approach to resolving high-degree vertices through count sketches, as we do, and additionally is not designed for dynamic graphs. A1 [40] is a dynamic attributed graph database that is in-memory and communicates over RDMA by building on FaRM [65]. A1 inherits FaRM's elasticity, but is limited in its scope to querying subgraphs and retrieve vertex and edge attributes; it does not support dynamic graph algorithms.

A demonstration of a system based on the Floe streaming engine, which is incremental-only and operates on snapshots, is given in [262].

Monarch [129] presents a model addressing geographically distributed graphs, where the latencies between separate partitions can be very high.

There are several high-performance shared-memory dynamic graph systems suitable for graphs that fit on a single node. RisGraph [83], GraphOne [146], and Aspen [59] are state-of-the-art shared memory systems. While they have impressive performance for both ingestion rates and analytics, as temporal graphs grow shared-memory cannot keep up. STINGER [67] and Llama [178] similarly are limited to shared-memory. GraphIn [230] extends the gather-apply-scatter model to the incremental case and develops a shared-memory implementation.

2.6.2 Dataflow Systems

Differential dataflow systems [188] are a general purpose strategy for handling incremental or dynamic computation. The Timely Dataflow system [198] includes a virtual clock which enables iteration processing in differential dataflows and subsequently supports graph processing. Apache Flink [41] provides both stream processing and batch processing in the same system, and includes a graph layer Gelly. Tornado [232] is a general purpose iterative computation framework that performs approximate computations until an accurate computation is requested. None of the above dataflow systems have addressed elasticity during computation, and require a re-configuration and re-deployment phase to add or remove resources.

2.7 Temporal Dense Regions

An important problem is determining dense regions in graphs that are evolving over time, instead of only finding dense regions for the immediate, most recent graph. There has been a significant amount of work on *temporal dense regions* that exactly solve this. As we show later, in Chapter 5, these definitions are all *vertex-focused*. They define a subgraph based on properties of the internal vertices, and then search in a temporal space to find the maximum

such subgraph.

Temporal dense regions can be broken down first by how density is defined: either as edge density, average degree density, the sum of weights, or other approaches such as cohesiveness. Second, temporal dense approaches have a *temporal* aspect which controls what target is pursued: the longest lasting region, observation of bursts, periodic behavior, and so on.

2.7.1 Edge Density

Using the ratio of number of edges in a subgraph to the total possible number of edges is the predominant approach for finding dense regions. The problem of finding the densest graph, an clique, is a classic NP hard problem. There have been numerous approaches to identifying temporal dense regions, which we outline here.

Wu et al. [264] uses the *number of edges* after projecting the temporal graph to a multi-graph as the second constraint for the dense subgraph, namely the h in (k, h)-cores. Recently, [18] studied (k, h)-core maintenance.

Li et al. [163] considers (θ, τ)-persistent k-cores, which are defined for a time interval I such that the k-core is present in any θ-length subinterval of I. The problem is shown to be NP-hard and provided heuristics first perform peeling and then search for such maximal time intervals. Li et al. [165] study (θ, τ)-continual k-cores from a *local perspective*, finding only cores that contain a query vertex, similar to a community search problem. Hung and Tseng [123] are defined similarly, proposing (l, k)-lasting cores which are maximal and last for time l.

Span-cores [95, 97], also known as (k, Δ)-cores, are defined with Δ as a time interval and such that each vertex in the (k, Δ)-core has a degree at least k for the entire interval. Note that if a vertex has a degree drop below k for any part of the interval, it cannot be used.

Qin et al. [206, 208] find both k-cores and k-cliques that occur periodically. Each

identified subgraph, which can be either a k-core or k-clique, must appear in a discretized graph at some constant p distance from a previous and next version. This problem is also NP hard for k-cliques, and the core variants are used to prune for clique finding.

Earlier work [4] builds a probabilistic pattern to match high density regions defined as pseudo-cliques. It uses min-hash to identify matches and takes a single pass over the stream. The goal is to find frequently occurring matches.

Lin et al. [167] studies diversified top-r lasting (k, σ)-cores. These are cores that last σ long, similar to other definitions, but the proposed mining objective finds the top-r cores that are *diversified*, that is they cover vertices and time as best as possible. This is also shown to be an NP-hard problem.

Yang et al. [268] looks at quasi-cliques that are present within a time range and [166] considers quasi-cliques that need to be present on average within a time range.

Yu et al. [269] finds historical k-cores and develops an index to quickly query them. Historical k-cores are k-cores that are present in some graph snapshot over a time range. This is not an inherently temporal core, but instead a static computation on a given snapshot, and can be viewed as lasting cores depending on the snapshot length.

Dai et al. [54] finds early bursting subgraphs which are k-cores, however it does not demand that the subgraph is around for some period of time.

2.7.2 Density as Average Degree

It is less common but can still be valuable to find dense regions where density is defined as the average degree for a subgraph, which is known as the densest subgraph problem. Unlike the ratio of edges to the possible number of edges, computing a maximal subgraph with the highest average degree is a polynomial time solvable problem.

Qin et al. [207] proposes an (l, δ)-maximal dense core which has an average degree at least δ in a time segment with length at least l.

Liu et al. [172, 173] looks for a densest longest subgraph that is present in a continuous

range. The proposed algorithm samples to get a match for their model, which is then used as the dense region.

Both Jethava and Beerenwinkel [130] and Semertzidis et al. [229] both find subgraphs that last a long period of time and are densest, defined either as the average degree or as the minimum degree.

Rozenshtein et al. [216] develops techniques to find the densest regions not only for average degree density but also generalized cover functions, which is an NP-hard problem.

2.7.3 Other Density Notions and Related Approaches

There are several other approaches that find dense regions with specialized definitions of density, or are related but not focused on dense regions.

In some approaches graphs have fixed nodes but temporally varying weights [31, 177]. They use the sum of edge weights (which are both positive and negative) as the subgraph score. This problem is NP-hard, and the computational approaches limit the number of temporal intervals to look at, given a temporal region T, from $|T|^2$ to $|T| \log |T|$ [31] or build upwards to consider only a small constant number of intervals.

Chu et al. [48] finds a dense region that quickly becomes dense, that is it is part of a burst of changes, and the dense region remains around for a time threshold. Density is a measure of cohesiveness of the temporal graph. This is shown to be an NP-hard problem.

Lahiri and Berger-Wolf [153] provide a polynomial time approach for finding periodic subgraphs. These are subgraphs that occur at a regular interval (with some jitter potentially present), but are not necessarily dense. Belth et al. [24] finds small motifs that are either persistent or bursty, and have a given frequency, but does not specialize in dense motifs.

2.7.4 Dense Regions and Community Detection

There is a significant literature developing effective and useful community detection algorithms for graph data [86], including temporal graphs [87], which differs in objectives from

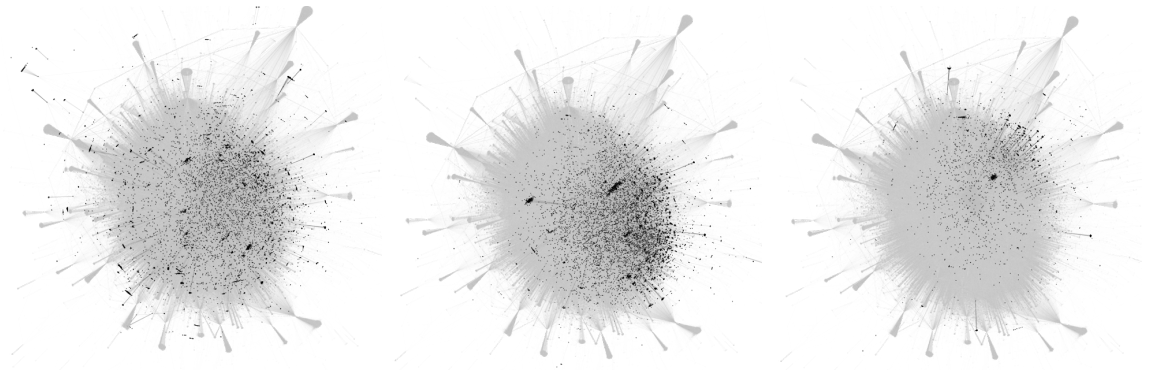


Figure 2.4: Running Louvain three times on a medium-sized graph. A dense region's community is highlighted. Each run of Louvain results in drastically different communities, showing the algorithm is unstable. Similar results occur for many graphs at this scale across many algorithms.

dense region tracking but shares some similarities. The objective is to label each vertex with one (or more, occasionally fewer) *communities* that it belongs to, where a community ideally represents a cohesive unit.

The most straightforward *temporal community detection* approach is to run community detection independently for each timestep (or graph snapshot), and then link the timesteps together [119, 16, 201]. This is similar at a high level to our core chain approach, described in Chapter 5. Various measures can be used to assist matching a community at time $t-1$ and time t, including many information theoretic measures such as normalized mutual information [185]. There are other approaches as well, such as FacetNet [168], that use approaches such as non-negative matrix or tensor factorization, however these approaches do not scale well to massive, sparse graphs. In order to understand why community detection based approaches are not well suited for identifying the hierarchical structure that is found from dense regions, we highlight an important problem with the stability of results [94].

Many community detection algorithms are designed, implemented, and validated on small or medium sized graphs—those with thousands to millions of edges. As a famous example, almost all community detection algorithms evaluate themselves against the karate club network [270] with 34 vertices (people) and 78 edges (friendships). The problem arises when the graphs increase in scale. As graphs scale to millions and billions of edges,

many algorithms can run (potentially after significant technical effort), but may fail to provide a useful result.

In Figure 2.4, one of the most popular community detection algorithms, Louvain [29], is run on a medium sized graph. A group of densely connected vertices is selected and the community corresponding to that group is highlighted. Even though no parameters are changed, multiple runs of Louvain produce completely different communities. We have found similar results with numerous variants of Louvain, including modern variants with patches such as Leiden [249].

An important reason to focus on dense regions is that such regions tend to be much more stable, and the approach of simply linking the points in the hierarchy with values that share the largest information theoretic measures or overlap does not apply well to the hierarchical nature.

CHAPTER 3

UNIFYING DENSE REGIONS THROUGH HYPERGRAPH CORES

Many real-world graphs are large, unstructured, and sparse. Within these sparse graphs are locally dense structures. Extracting dense subgraphs from such graphs has proven useful for a variety tasks including deriving news stories from blogs [13], finding communities in websites [148, 64], identifying link spam websites [103], and uncovering motifs in DNA [88]. Not only are dense subgraphs inherently useful, they are helpful as kernels in graph visualization [10] and optimizing computation [132]. As such, dense subgraph extraction is a fundamental problem in graph mining.

Finding the largest densest subgraph, or maximum clique, is a classic NP-hard problem [138], and finding clique-like objects remains hard [39, 52]. Instead, dense subgraph extraction has shifted towards the notion of *cores* of a graph [228, 183]. The k-core is a maximal connected subgraph where every vertex has an induced degree at least k. Cores in a graph produce *hierarchies* of dense regions in a laminar family. Importantly, cores can be computed in linear time efficiently.

Unfortunately, on some graphs k-core hierarchies are funnel-shaped around a single large core and fail to uncover other useful, deep structure in graphs [122, 50, 223]. As an example, funnel-shaped hierarchies fail to uncover separate similarly dense regions, track vertex movement between such regions, or measure non-trivial changes in the hierarchy's structure over time. As such, there have been extensions to and generalizations of cores hoping to improve the resulting quality, including adding constraints [271], using weights [71], and abstracting to edges [50] and other clique-inclusions [223]. For each of these derivative targets, custom algorithms, data structures, and techniques are developed.

For static graphs, the algorithms tend to be simple and easy to implement and maintain. However, this becomes a significant problem with dynamic graphs, where edges contin-

uously change as, for example, real-world data continues to stream. In this case, core maintenance algorithms are complex and difficult to develop [224, 164, 274, 121]. Maintenance for corresponding extensions, while heavily relying on concepts from k-cores, are similarly involved [122, 273, 271]. Dynamic algorithms, or *maintenance* algorithms, are increasingly important as data rates increase and interactive or even real time (low latency) queries are desired. Consider detecting lateral phishing attacks [117] in enterprise email systems: a vertex joining many dense regions indicates that a user is reaching out to multiple separate cohesive groups. Today's enterprises see tens of thousands of emails daily and a low detection latency (along with a low false positive rate) could allow an attack to be stopped. A static algorithm running a few times per day would be insufficient but a maintenance algorithm with a latency of tens of seconds would be enough.

For this reason, and their practical importance, we consider maintenance algorithms. We focus on the generalization of graph cores to nuclei [223], which includes truss communities. We propose a unifying framework that reduces any nuclei computation to a core computation on hypergraphs which is very closely related to computing graph cores. This importantly allows for any progress in core maintenance algorithms to directly and effortlessly impact maintenance algorithms for trusses and its generalization to nuclei.

Our approach splits the problem into two parts. First, we maintain a hypergraph that contains components of nuclei. We do this by identifying whether the addition or removal of an edge would, respectively, complete or break a small, parameter-sized clique (typically a triangle or 4-clique). Second, we maintain k-cores in the hypergraph using standard k-core maintenance algorithms. Our approach and implementations are general for arbitrary nuclei.

To demonstrate the application of this novel approach and retention of efficiency of algorithms built for graphs when applied to hypergraphs, we introduce traversal [224] and order [274] hypergraph core maintenance algorithms. Our traversal algorithm, configured to solve trusses, has similar performance to the best publicly available truss maintenance

implementation DyTruss [122] when clustering coefficients are high and outperforms it otherwise. This is expected, as DyTruss is derived from traversal-based k-core algorithms. Our order algorithm always outperforms traversal and DyTruss, similar to [274] outperforming traversal for k-cores.

Furthermore, we show that our approach provides a useful maintenance algorithm for nuclei. Over a variety of real-world graphs, we return queries before the cost of computing nuclei from scratch, even for complex nuclei.

We have the following contributions:

- We prove in the locally persistent model that any nucleus maintenance algorithm is unbounded, informing our complexity analysis.
- We relate cores and nuclei by proving computing nucleus decompositions is equivalent to finding k-cores in an (r, s)-hypergraph H.
- We establish properties crucial to maintenance on hypergraph k-cores and provide two k-core hypergraph maintenance algorithms: traversal and order.
- We provide a fully dynamic nucleus algorithm, using small clique enumeration and k-core maintenance.
- We perform an experimental evaluation and demonstrate our implementation's applicability for providing low latency results on rapidly changing, moderately sized real-world graphs and that developing k-core hypergraph maintenance algorithms immediately apply to core, truss, and other nucleus maintenance.

The remainder of this chapter is organized as follows. § 3.1 studies the complexity of nucleus maintenance. § 3.2 equates nucleus decompositions and hypergraph k-cores and develops core maintenance results in hypergraphs. § 3.3 introduces our new H maintenance algorithm and hypergraph changes to k-core graph algorithms. § 3.4 presents experimental results and discussions and § 3.5 provides a discussion on the results.

3.1 Computational Complexity of Updates

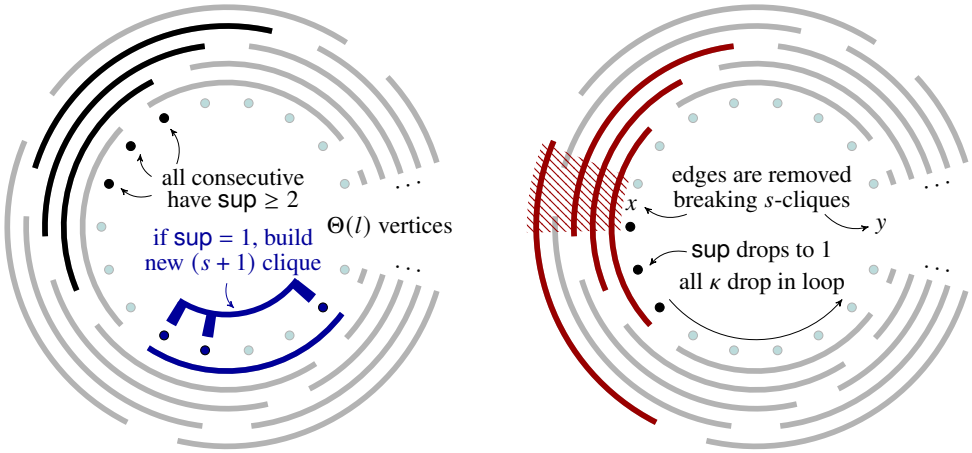
Dynamic algorithms are used when latency is more important than throughput or efficiency. Can we understand how much lower the latency will be? Such algorithms are not well suited for standard worst-case complexity; there can be a large range of useful algorithms $\mathcal{A}_\Delta$ with $O(T_{\mathcal{A}_\Delta}) = O(T_{\mathcal{A}})$. The locally persistent model [211, 8] was developed to explore a dynamic graph algorithm's complexity in terms of the fundamentally necessary changes to the graph as determined by $\mathcal{A}$ and recently extended to consider batches [77]. Algorithms in this model are constrained to store only local data per-vertex and not include any auxiliary or global data structures. A problem is bounded in this model if for some $\mathcal{A}_\Delta$ there exists a polynomial f and constant c such that for any stream $\mathcal{S}$ and time range $i < j$, $\mathcal{A}_\Delta(G^{(i)}, \mathcal{A}(G^{(i)}), \mathcal{S}_i^j)$ runs in $O\left(f(|X|_c)\right)$, where for $|X|_c$ is the c-hop neighborhood around vertices that have their local data changed between time i and j.

A bounded algorithm in this model may have $o(|E|)$ runtime complexity when only some of the graph changes, whereas an unbounded algorithm cannot. We study boundedness in this model because a bounded algorithm can then provide asymptotically less work. For unbounded problems, we can frame complexity in problem-specific models [77, 273] or study algorithms' empirical latency distributions over representative datasets.

Zhang et al. recently proved that under the locally persistent model it is not possible to bound edge insertions for computing k-cores and k-trusses [273]. We demonstrate a more general result, that the unboundedness holds for all dynamic (r, s) nucleus decompositions. We prove this by building a specific graph such that any dynamic algorithm must visit the whole graph on certain inputs, even when no change will result.

Theorem 3.1. *Let $\mathcal{A}_\Delta$ be a locally persistent (r, s)-nucleus maintenance algorithm, with $r < s$. Then $\mathcal{A}_\Delta$ is unbounded.*

Proof. This is a proof by contradiction. We will carefully construct a graph G such that the individual insertion of two far apart edges will not change any κ values. However, if



(a) The graph before s-cliques are broken. (b) The finished unbounded graph.

Figure 3.1: An example unbounded graph constructed for $(3, 5)$ nuclei where dots are vertices and arcs show an s-clique among vertices under it. In (a), the graph with $\kappa = 2$ is built. All consecutive r-cliques have their support from the ring, and all other r-cliques have new $(s + 1)$-cliques built to maintain their support. In (b), two edges are removed arbitrarily far apart from each other. The r-cliques in the ring will drop their κ values as they lose their support. Adding in just one edge will not change κ; only adding both in will recover κ.

both edges are inserted, κ values will change. This forces *any* algorithm on either insertion to visit an arbitrarily large portion of the graph, and hence will show that $\mathcal{A}_\Delta$ must be unbounded.

Assume that $\mathcal{A}_\Delta$ is bounded. Construct G as follows. Let l be an integer and $s > r > 0$ be integers. Create $2l + k$ different s-cliques overlapped on one vertex, without letting any vertex be overlapped more than s times, for some small k. Next, connect the overlapping s-cliques back to the beginning, forming a loop with some vertices having support of 2. Let x, y be two farthest apart edges with a support of 2. For each r-clique with a support of 1, create a new $(s + 1)$-clique it is part of. Finally, remove x and y. An example of the construction for $(3, 5)$ nuclei is shown in Figure 3.1. This overlapping construction tool is illustrated in Figure 3.2.

Observe that the r-cliques with an original support of 1 now have a κ value of at least 2, and those in the original loop, which has been broken, now have a κ value of 1.

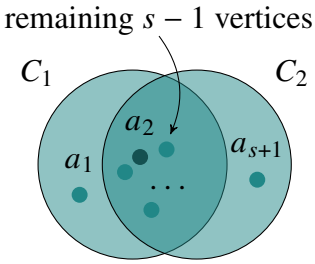


Figure 3.2: Construction tool in Theorem 3.1. C_1 and C_2 are s-cliques sharing all but a_1, a_{s+1}, and $\mathsf{sup}(a_2) \geq 2$.

Let R be the set of r-cliques whose κ value changes, $|\mathsf{R}|_c$ be the c-hop neighborhood around R, and $\mathsf{tr}(\mathcal{A})$ denote vertices visited by $\mathcal{A}$. Now, as $\mathcal{A}_\Delta$ is bounded by assumption, there exists a polynomial function f and constant c such that for all graph changes S, $\mathcal{A}_\Delta(G, \kappa, S) \in O(f(|\mathsf{R}|_c))$. Note that if only x or y is added to G then no κ values will change. So, we know that $\mathsf{R} = \emptyset$ for both $(G, \kappa, (\langle x, + \rangle))$ and $(G, \kappa, (\langle y, + \rangle))$, and so $O(f(|\mathsf{R}|_c)) = O(1)$ and hence for $S \in \{(\langle x, + \rangle), (\langle y, + \rangle)\}$, $|\mathsf{tr}(\mathcal{A}_\Delta(G, \kappa, S))| \in O(1)$.

Now, consider $S = (\langle y, + \rangle)$. If x has been inserted already, we have $\kappa \geq 2$ for all r-cliques in G. Let z be an r-clique in the middle of the loop, that is $\Theta(l)$ away from x and y, that has a κ change. Recall locally persistent algorithms can only make changes based on a traversal of the graph, so $\mathcal{A}_\Delta$ has no way of knowing whether the graph is G or $G \cup \{x\}$ without visiting x's endpoints. Yet, z must change when it is $G \cup \{x\}$, but not when the input graph is G. Also, z must be visited with both $G \cup \{x\}$ and $G \cup \{y\}$, as applying either individually has the same impact to κ. So,

$$|\mathsf{tr}(\mathcal{A}_\Delta(G, \kappa, (\langle x, + \rangle)))| + |\mathsf{tr}(\mathcal{A}_\Delta(G, \kappa, (\langle y, + \rangle)))| = \Theta(l),$$

which is a contradiction and thus $\mathcal{A}_\Delta$ cannot be bounded. □

Theorem 3.2. *Let $\mathcal{A}^-_\Delta$ be a locally persistent (r, s)-nucleus decremental algorithm, with $r < s$. Then $\mathcal{A}^-_\Delta$ is bounded.*

Proof. (Sketch) Consider our algorithm, presented in § 3.3. Note that it is locally persistent when using a locally persistent hypergraph k-core algorithm, such as a traversal based

algorithm. Boundedness follows by applying Theorem 3.3, presented next. □

Given Theorem 3.1, we cannot expect to find a bounded algorithm and under standard models our worst-case runtime complexity will be the same as a full, static computation. Note that the structure constructed in this proof is not likely to exist in real-world graphs, and so we instead focus on empirically demonstrating the behavior of algorithms and work to reduce variability.

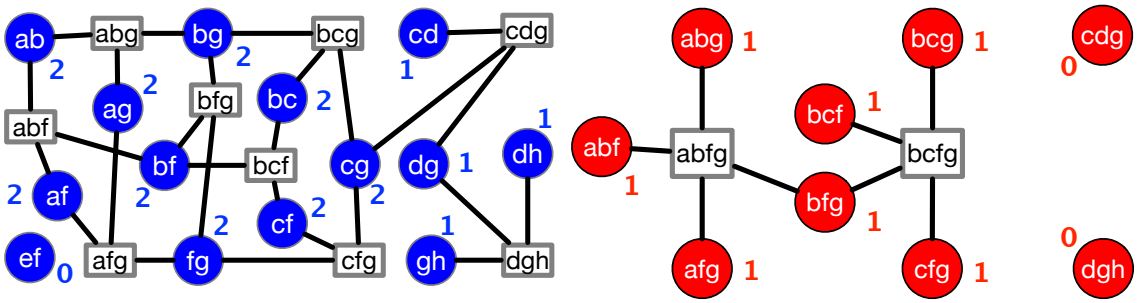
3.2 Nuclei and Hypercores

In this section we establish that nucleus decomposition can be reduced to hypergraph k-core computation. We first introduce the (r, s)-hypergraph of a graph G. Let $H = (X, Y)$, where each vertex $v \in X$ is an r-clique in G, and each hyperedge $e \in Y$ is an s-clique in G. An r-clique v is present in a hyperedge e, i.e., $v \in e \in Y$, when that r-clique is contained in the corresponding s-clique in G. The degree of each vertex in H is the support of the corresponding r-clique in G.

An example showing how H reflects r- and s-cliques is in Figure 3.3. Figure 3.3a shows a $(2, 3)$-hypergraph where each edge of the original graph is represented as a vertex (circle) and each triangle is a hyperedge (rectangle). Similarly Figure 3.3b shows a $(3, 4)$-hypergraph where each triangle of the original graph is a vertex and each 4-clique a hyperedge.

Establishing that computing the hypergraph k-core is equivalent to computing the (r, s) nucleus decomposition in G is important, as it allows the computational problem to be split into two: first, H can be maintained by adding and removing hyperedges as necessary when s-cliques are formed and destroyed; second, k-cores in H can be maintained.

Theorem 3.3. *Let G be a simple undirected graph and $r < s$. Let H be the (r, s)-hypergraph corresponding to G. There is a one-to-one correspondence between a k-(r, s) nucleus in G and a k-core in H.*



(a) $(2,3)$-hypergraph (b) $(3,4)$-hypergraph

Figure 3.3: $(2,3)$- and $(3,4)$-hypergraphs of the graph displayed in Figure 2.1 with $(2,3)$- and $(3,4)$-nucleus decomposition values. Vertices (circles) and hyperedges (rectangles) are labeled for presentation.

Proof. Let $C \subseteq G$ be a k-(r,s) nucleus in G. Then, C is a maximal set of edges of s-cliques such that for each r-clique u in an s-clique S, u has an S-degree at least k and is S-connected. Let K be the set of r-cliques that are in C. Let $S_1, \ldots, S_l$ be the s-cliques that u participates in. In H, u will participate in the edges $S_1, \ldots, S_l$. Since $l \geq k$, we have $\deg_H(u) \geq k$. Also for all r-cliques v in C, there is a path $u, u_1, u_2, \ldots, u_m, v$ such that $\{u, u_1\} \in S'_1$, $\{u_m, v\} \in S'_{m-1}$, and $\{u_i, u_{i-1}\} \in S'_i$ for all $0 < i < m$, for some S'_i. So, for every $u, v \in K$, we have connectivity.

Now, we need to show that K is maximal. Suppose that $K' \supset K$ is a k-core in H. Recall that k-cores in hypergraphs require *every* vertex u in each edge to have degree at least k, so for all $e \in E(H[K'])$ (edges from H' induced from the vertices K'), we have that for $u \in e$, $\deg_{H[K']} \geq k$. Let C' consist of the edges in G corresponding to each s-clique in $E(H[K'])$. Then, $C' \supset C$, and C' is a k-(r,s) nucleus as H is constructed from r and s-cliques, and K' is connected, which is a contradiction that C is maximal.

A similar argument shows that finding a k-core K in H will produce a k-(r,s) nucleus in G. □

After finding H, the k-(r,s) nucleus can be computed by determining the k-core in H, by applying Theorem 3.3. The importance of this result is that we can apply results from k-core maintenance after only minor extensions.

For the following lemmas let $H = (X, Y)$ be a hypergraph. Let y be a hyperedge with vertices $\{d_1, \ldots, d_s\}$, which will be either added to or removed from H. Let d_i be one of the vertices in y with a minimum κ value, so $\kappa[d_i] \leq \kappa[d_j]$ for all $1 \leq j \leq s$. Let κ' represent the nucleus decomposition values corresponding to $H' = (X, Y \cup \{y\})$, and κ represent the original.

Lemma 3.1. *For all $x \in X$, $|\kappa'[x] - \kappa[x]| \leq 1$.*

Lemma 3.2. *If $\kappa[d_j] \neq \kappa[d_i]$ then $\kappa'[d_j] = \kappa[d_j]$.*

Lemma 3.3. *$H'[\{x \in X : \kappa'[x] \neq \kappa[x]\}]$ is connected.*

Lemma 3.4. *Let $X = H[\{x \in X : \kappa[x] = \kappa[d_i]\}]$. Only vertices c in the same connected component as d_i in X* may *have $\kappa'[c] \neq \kappa[c]$.*

The proofs follow well-known subcore result proofs, i.e., Theorems 1–4 [224], noting that hyperedge changes only impact the degree by one and hypergraph k-cores are defined by minimum degree of all vertices in a hyperedge. These lemmas are crucial for converting graph core algorithms to hypergraph algorithms: they show that on any edge change, the κ values can only change by one, and the change will occur within a *subcore*, a connected region of the hypergraph where all vertices have the same initial κ value. This enables ignoring certain edges, reducing the total work.

3.3 Maintaining Nucleus Decompositions

The goal is to maintain the nucleus decomposition values, κ, under the dynamic graph stream, $\mathcal{S}$. We split this problem into two parts: first, we maintain the (r, s)-hypergraph H, an intermediate structure that represents r- and s-cliques in G; next, we use H to maintain κ, which are hypergraph coreness values.

3.3.1 Maintaining H on a Changing Graph

The algorithm proceeds as follows. Consider each edge in the stream and determine whether that edge creates or deletes s-cliques, by enumerating all s-cliques that involve both endpoints. For each creation or deletion, all internal r-cliques will be impacted; their S-degree will change by one and a hyperedge will be created or removed. When the change is an edge deletion, the graph is updated after enumerating impacted hyperedges, capturing any s-cliques it was part of. For insertions, the update is before.

Correctness Suppose y is an s-clique. Under any ordering, there will always be a final edge that creates y; before that edge, no internal r-cliques are connected through y. After that edge, all internal r-cliques have a connection. The same applies to the first edge removal of y: post removal the corresponding hyperedge also no longer exists.

Complexity We consider the worst case complexity for processing any edge update. Consider the update $\langle \{u, v\}, \pm \rangle$ where $C_{u,v}$ denotes the number of common neighbors between u and v. Note $C_{u,v} = O(|V|)$. The number of potential s-cliques we enumerate is $\binom{C_{u,v}}{s-2}$ each of which contain $\binom{s}{r}$ r-cliques, resulting in a runtime per edge of

$$O\left(T_G + T_{CN} + \binom{C_{u,v}}{s-2}\left(\binom{s-2}{2}T_C + \binom{s}{r}\left(T_H + O(f)\right)\right)\right),$$

where T_G is the complexity of updating the graph, T_{CN} of finding common neighbors, T_C of querying an edge, and T_H of updating the hypergraph. The runtime is dependent on the existence of large cliques in the graph, and it is an open problem to find any nucleus decomposition strategy–maintenance or otherwise–to handle these cases efficiently. Note that for low nuclei such as cores, trusses, $(2, 4)$, and $(3, 4)$ our algorithms and implementations are efficient even with large cliques and the complexity matches that of core, truss, and nucleus decompositions. By Theorem 3.1, we cannot expect to do better.

This approach can be extended to support arbitrary sized batches, by processing dele-

tions first and insertions second.

3.3.2 Computing κ

To determine the final nucleus decomposition values using H, we can use any hypergraph k-core maintenance algorithm. Applying Theorem 3.3, the resulting cores will be nuclei. Finally, Lemma 3.1-Lemma 3.4 provide the connection between hypergraph cores and graph cores necessary enabling graph core algorithms to be directly used.

For our implementation, building on the two state-of-the-art incremental and decremental core maintenance algorithms, namely traversal [224] and order-based [274], we developed two new hypergraph k-core maintenance algorithms: traversal and order. At a high level, traversal begins at the inserted edge and performs a depth-first search. It eliminates vertices from consideration that are not in the subcore or cannot have their coreness change. The decisions are made by evaluating the maximum possible degree. order simulates a sequential peeling process and maintains an order in which vertices could be peeled, using the order to skip vertices. The order is maintained based on subcores.

We only describe in detail algorithm changes to operate on hypergraphs. We use the notation from [224] and [274], respectively. In each algorithm, the decisions to follow an edge or not needs to change to consider *multiple neighbors* in a hyperedge.

First, we introduce the *hypergraph maximum-core degree* (MCD) and *hypergraph pure-core degree* (PCD). PCD is used for the two-hop traversal variant [224], and MCD is used in both the traversal algorithm and the decremental order-based algorithm [274].

Definition 3.1 (Hypergraph MCD)**.** *Given H and vertex u,*
$\mathsf{MCD}(u) = |\{e \in \Gamma_H(u) : \forall v \in e \setminus \{u\}, \kappa[v] \geq \kappa[u]\}|$.

Definition 3.2 (Hypergraph PCD)**.** $\mathsf{PCD}(u) = |\{e \in \Gamma_H(u) : \forall v \in e \setminus \{u\}, \kappa[v] > \kappa[u] \vee (\kappa[u] = \kappa[v] \wedge \mathsf{MCD}(v) > \kappa[u])\}|$.

For traversal, MCD and PCD are maintained by exploring the one and two hop neigh-

Table 3.1: The graphs used for our experiments.

	Units: Millions			# Cliques	
Graph	Type	$\|V\|$	$\|E\|$	3	4
DBLP [162]	temporal	1.82	8.34	27.1	600
WikiConflict [37]	temporal	0.12	2.03	13.8	63.3
Google [107]	static	0.88	4.32	13.1	39.9
YouTube [191]	temporal	3.22	9.38	10.5	29.1
Facebook [255]	temporal	0.06	0.82	3.39	13.3
Gowalla [47]	static	0.20	0.95	2.13	6.09
Patents [158]	static	3.77	16.5	4.23	3.50

borhoods around an edge. Additional checks are added to prevent repeatedly evicting hyperedges and traversal decisions remain based on the hypergraph MCD and PCD.

For order, each hyperedge *internally* has an order. The degree counters ($\deg^+$, $\deg^*$) only track the *first* internal vertex per hyperedge and with an order movement (third case in [274]) the vertices in a hyperedge are all re-ordered preserving their internal order.

Our approach does not preclude using fully dynamic batch algorithms, although for simplicity they are not considered here. Hypergraph core batches can be applied after all hyperedge changes have been made, either for a batch of edge updates or for a single edge update. In particular, the hypergraph core algorithms operate on a *hyperedge stream*, implicitly constructed while maintaining H.

3.4 Experiments and Results

In this section we describe our experiments and results.

3.4.1 Experiments

Datasets We evaluated on real-world graphs, shown in Table 3.1. We evaluated using four temporal graphs, which consist of ordered insertions, and three static graphs, each randomly ordered to simulate a stream. We remove multiedges and self loops and treat all graphs as unweighted and undirected. Our graphs represent social networks at the scale

Table 3.2: Edges before reaching the static runtime.

	truss		$(3,4)$	
Graph	Static (s)	Edges	Static (s)	Edges
DBLP	28.4	137k	1223.7	182k
WikiConflict	19.4	41.8k	301.3	85.2k
Google	11.4	9.6k	56.2	5.51k
YouTube	111.9	529k	692.7	453k
Facebook	3.6	12.8k	26.6	12.1k
Gowalla	3.6	6.86k	20.4	4.03k
Patents	36.5	452k	38.8	209k

existing static nucleus algorithms are run.

Focus on Higher-Order Nuclei Our experiments and datasets focus on approaches that extend to higher nuclei, such as $(2,4)$ and $(3,4)$. These have been shown to provide non-linear hierarchies when cores, and trusses only provide a single, funnel-shaped densest region hierarchy [223]. Extending any nucleus approach efficiently to $(r,5)$, $(r,6)$, and higher will likely involve approximations. Future work could place approximation-derived weighted hyperedges into our unified framework. We do not show deletions as they are bounded (Theorem 3.2.)

Implementation We implemented our algorithms in C++17 and compiled with GCC 10.1.0 and `-O3`. We used Abseil flat hash maps with vectors to store both the dynamic graph and hypergraph. All data points are the average of a run of five independent trials and all times include memory allocations. Our experiments were run on Intel Xeon E5-2683 v4 CPUs with 512 GB main memory and dual sockets. Memory is measured using `malloc_count`. We implemented order with an order-statistic treap and a binary heap. All experiments use order, unless explicitly testing traversal. For baselines, we use `nd` [223] and DyTruss [122]. [273] did not have an available implementation.

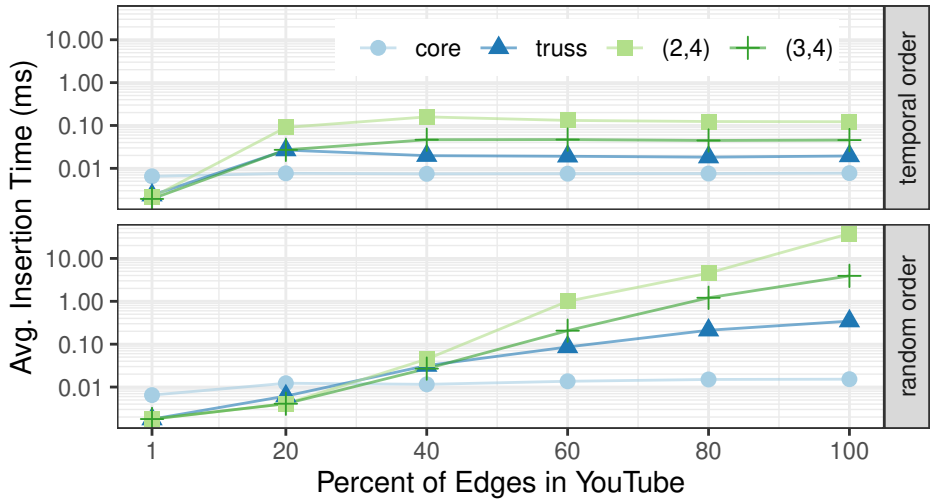


Figure 3.4: Average insertion latency of `YouTube` at points in time ordered either temporally or randomly. As is standard practice, the average per-edge runtime is calculated from batches based on various percentages of the full stream data. The graph is pre-loaded to $|E| \times \% - 10\text{k}$ edges, and then the average per-edge time for the next 10k edges is shown.

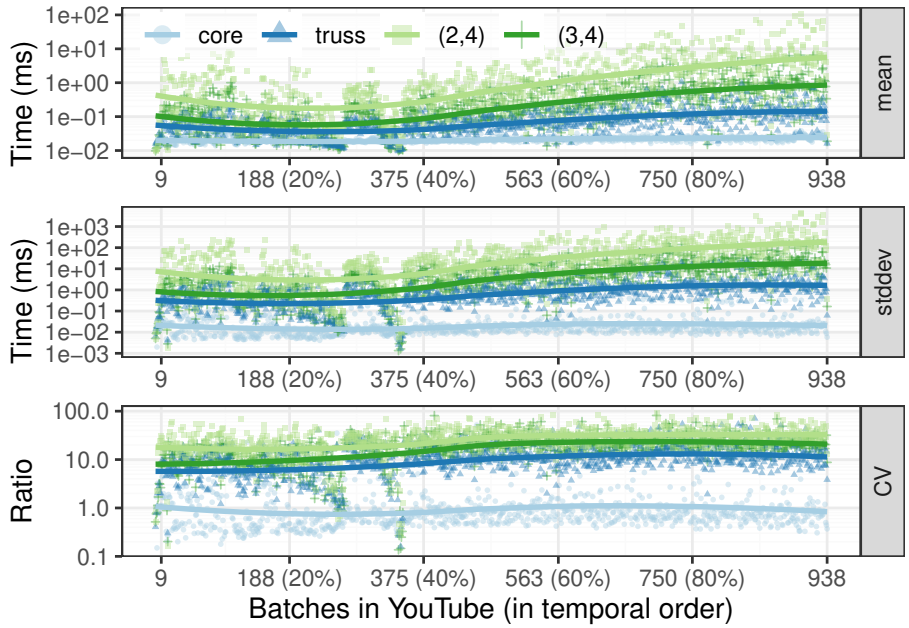


Figure 3.5: Standard statistics for runtimes within each non-overlapping 10k batch in `YouTube`, ordered temporally. Lines are smoothing curves of *per-edge* means, standard deviations, and coefficients of variance *within each batch*. Batches start with edges $\mathcal{S}_1^{10\text{k}}$ and end with edges $\mathcal{S}_{|E|-10\text{k}}^{|E|}$. Also, Figure 3.5 shows all the batches as run through the system, not just 10k edges at particular percentages of the full stream. Compare this with the *temporal order* in Figure 3.4 (top). Unlike choosing only a few arbitrary 10k batches, using all the batches shows a clear upward trend for all nuclei.

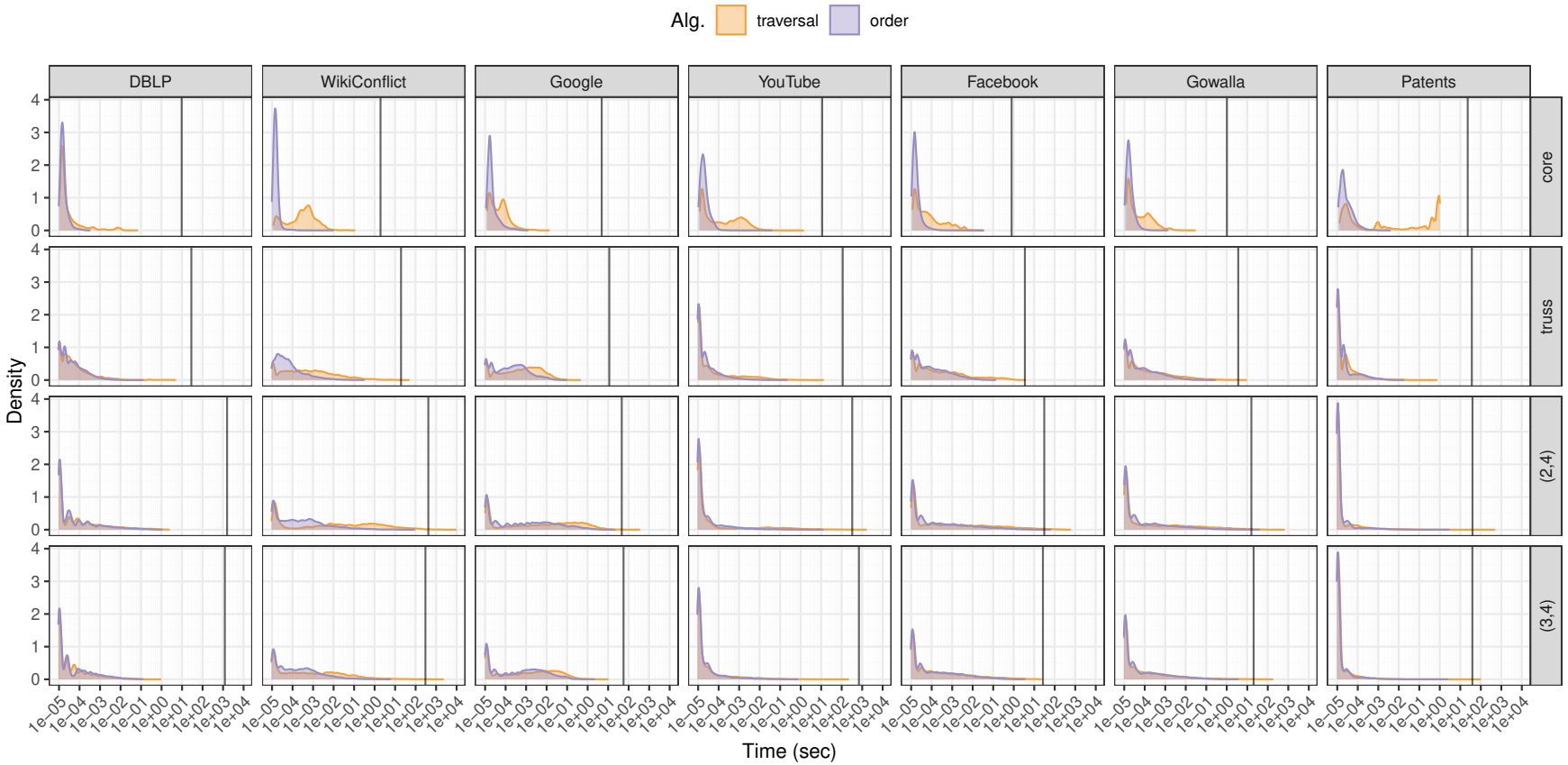


Figure 3.6: Distributions showing variability and latency. The black line indicates fastest static implementation. In each case, order is a better choice than traversal, as the tails for traversal extend farther and the mean is lower for order. For all algorithms the tail is long, as expected from Theorem 3.1. Future work on cores, trusses, or any nuclei maintenance algorithm should explicitly focus on reducing long tails.

Experiments and goals We have two experimental goals.

Goal 3.1. *Experimentally determine whether our algorithm has low enough latency with high enough throughput for real use cases.*

Goal 3.2. *Determine whether our approach removes the need to develop specialized nuclei maintenance algorithms.*

First, we perform four experiments to address Goal 3.1.

Experiment 3.1. *We measure the average per-edge runtime for inserting at various points in the stream, a common evaluation for dynamic algorithms. We insert* $10k$ *edges. See Figures 3.4–3.5.*

Experiment 3.2. *We measure how many edges are processed, starting from the last edge inserted into a graph and working backwards, before the latency of our algorithm crosses the run-time of recomputing from scratch. See Table 3.2.*

Experiment 3.3. *We report per-edge runtime distributions and compare to re-computing from scratch. See Figure 3.6.*

Experiment 3.4. *We measure the memory used by our algorithm, both when maintaining the entire hypergraph H in memory and when computing it on-demand as a* virtual hypergraph*. See Figure 3.7. We report the computational overhead of this approach in Figure 3.8.*

To address Goal 3.2, we introduce two more experiments.

Experiment 3.5. *We measure the fraction of time spent maintaining k-cores, maintaining H, and other processing.*

Experiment 3.6. *We compare our algorithm's average runtimes with both* traversal *and* order $(2,3)$*-nucleus along with the publicly released truss maintenance implementation,* DyTruss *[122].* DyTruss *is only released as a binary, and instead of running it edge-by-edge we ran it with* 10 *edges at a time to mitigate its startup inefficiencies.*

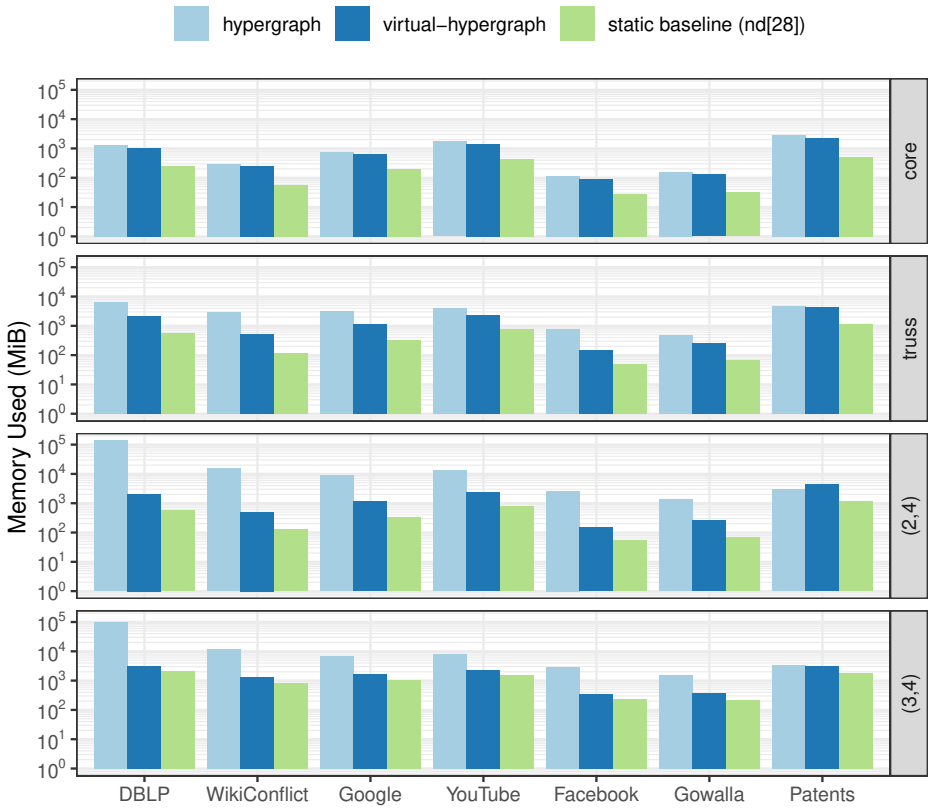


Figure 3.7: Total memory use is well within a modern server's memory. We store graphs in hash tables with 64-bit vertex IDs. Future work may benefit from memory optimizations.

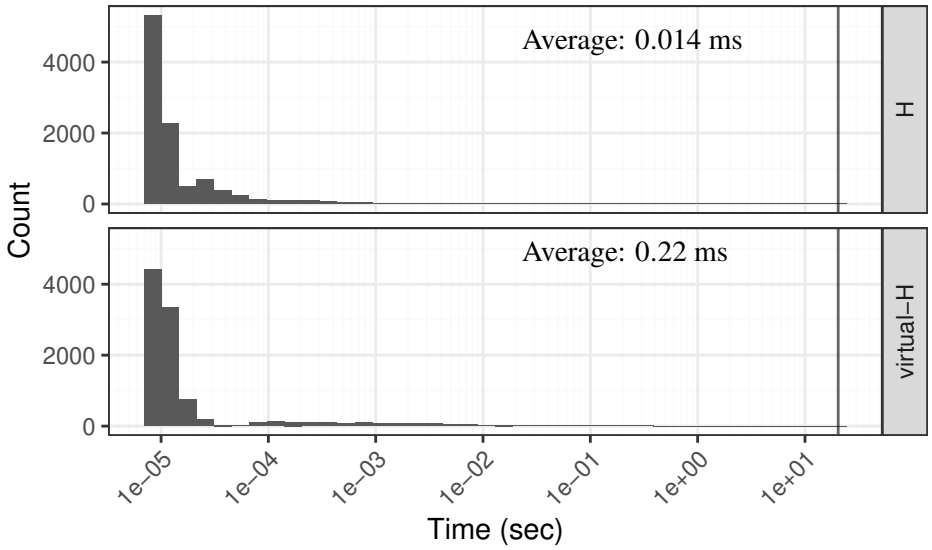


Figure 3.8: Virtual-H shows the latency impact on Gowalla of not storing H and computing its edges on-demand for a memory intensive case with $(3,4)$ nuclei. The tail becomes longer yet many results can still be returned quickly.

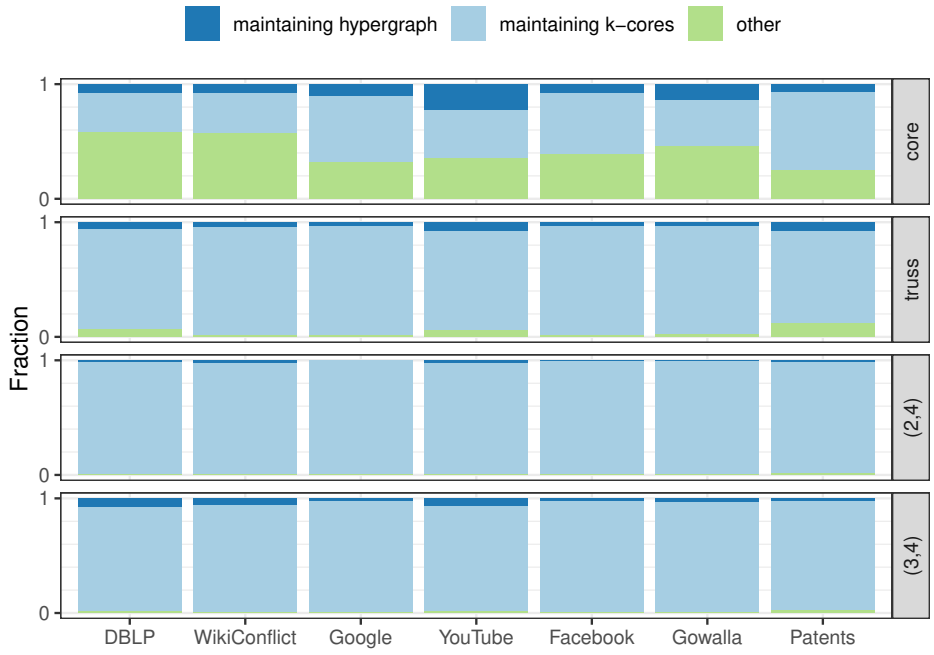


Figure 3.9: Almost all time is spent maintaining the hypercore instead of maintaining the hypergraph itself, showing an algorithmic focus on cores is important.

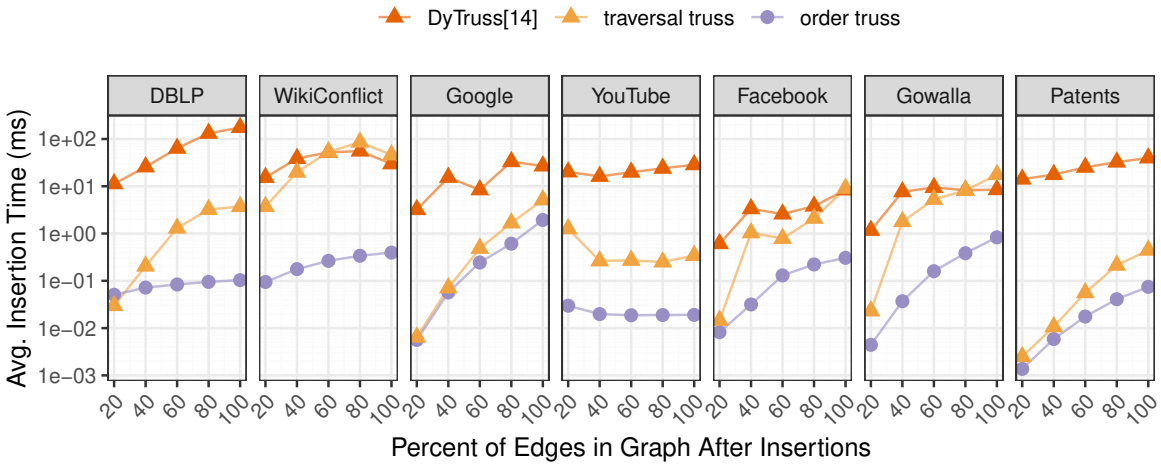


Figure 3.10: Comparison of two variants of our unified framework (order and traversal) against DyTruss [122] for trusses. Triangles indicate traversal-based, circles order-based.

3.4.2 Experimental Results

We report on Experiments 3.1-3.4, addressing Goal 3.1. From Figure 3.4–3.6, at all scales and for every edge and graph, our algorithm returns with a latency lower than the baseline of re-computing from scratch. We show temporally ordered graphs have a similar scalability with with randomly ordered graphs, similar to results on k-cores [274]. Furthermore, we show it is better to report distributions for *all* batches (Figure 3.5); randomly ordering may miss inherent locality (dips between 20% and 40% in Figure 3.5), and a few sample points may miss trends. We show the high variance is not only within batches but among them, with very *infrequent* but impactful long tails. We achieve maximum improvements of between 8.2×10^4 and 1.9×10^8 and average improvements between 4.3×10^4 and 8.7×10^7 across the tested graphs. For $(3,4)$ nuclei, the average improvement is 1.7×10^7. Table 3.2 shows our algorithm sustains edge-per-second rates that are likely beyond real-world update rates for medium-sized graphs, such as corporate email graphs. Furthermore, we are computationally bounded for high r and s nuclei, and so our memory consumption, shown in Figure 3.7 (without a virtual hypergraph) is well within a standard server's memory. As such, we identify this approach is suitable for use as a maintenance algorithm.

The virtual hypergraph memory optimization enables a tradeoff between low latencies and memory constraints: as shown in Figure 3.8, it is around 15× slower on average for `Gowalla`, has worse long tails, yet stays below re-computing and uses asymptotically tight memory. Hybrid approaches may be an avenue for scaling to very large graphs—especially if coupled with approximations.

Importantly, we show the distributions are not normal and as such we demonstrate reporting only the average is inadequate. This holds even for cores and truss communities. Future algorithms should address long tails instead of solely focusing on averages.

Next, we report on Experiments 3.5-3.6, addressing Goal 3.2. One key outcome of our algorithm is that it reduces the computation of nuclei, including trusses, to computing k-cores on H. For this to be useful, we need two properties: first, H maintenance itself

cannot add a significant overhead. Figure 3.9 shows the runtime is dominated by maintaining k-cores. Second, algorithmic improvements for cores need to apply to other nuclei. Figure 3.10 shows that traversal and the traversal-based DyTruss [122] have similar characteristics, for example consider `WikiConflict`. Where they are dissimilar the $(2,3)$ hypergraph is much smaller (see Table 3.1).

Our results demonstrate that algorithmic improvements to core maintenance directly carry through our nucleus decomposition approach, removing the need for manually porting results between cores and trusses.

3.4.3 Related Work and Future Directions

To the best of our knowledge, we provide the first nucleus maintenance algorithm on dynamic streams. Two special cases, computing k-cores and k-trusses, have seen several maintenance algorithms: [224, 164] reduces work by comparing with existing core numbers and [274] maintains a decomposition order. [121] operates in the parallel batch setting and identifies independent edges to process concurrently using specialized trees. [122] introduces k-truss communities and provides a traversal based incremental algorithm. [273] presents an ordering based batch truss maintenance algorithm, shows insertions are unbounded, and develops an analysis framework that bounds them. Their implementation is unavailable, however we outperform [273]'s reported runtimes, on the same processor, with our order implementation on `soc-LiveJournal1` by over 93×.

[226, 171] address cores on bipartite graphs, or hypergraphs where hyperedges can be broken. [243] presents an approximate hypergraph k-core maintenance algorithm. Our H maintenance has similarities to the maximal clique approach in [55]. Future work could leverage this within our framework by generalizing nuclei with other dense structures, potentially with sampling or sparsification.

3.5 Summary

In this chapter, we studied the problem of maintaining nucleus decompositions on dynamically changing graphs. We proved any locally persistent dynamic algorithm is unbounded, meaning worst-case runtimes may equal full re-computation even if the output does not change. We established an equivalence between hypergraph k-cores and nucleus decomposition, forming the theoretical foundation of our unifying framework. Our approach splits the problem into two: maintaining a special hypergraph and then maintaining k-cores on it. We implemented our unified framework using order and traversal k-core approaches for arbitrary (r, s) values. We can then maintain nuclei with lower latencies than re-computing from scratch with sufficient throughput to drive real-world uses on medium-sized graphs. Our work enables interactive use of significantly more interesting and useful nuclei than only cores or trusses and allows progress in core maintenance to directly impact all nuclei.

CHAPTER 4

FROM CORENESS TO CORES

Many practically important graphs from web data, social networks, and related fields are both large and continuously changing. The problem of maintaining core *decompositions* on graphs has been well studied [164, 225, 274, 273]. Existing approaches run in linear time in the size of the graph, which is theoretically optimal [273], and on many real-world graphs they maintain decompositions within milliseconds after edge changes. So, is the problem solved?

Unfortunately, these approaches only address half of the problem of returning a k-core[221]. k-cores are originally defined as *connected* subgraphs [228]. All of the application examples referenced above rely on or use connectivity. A core decomposition, on the other hand, provides *coreness* values for every vertex: that is, the largest value such that a vertex is in a k-core, but not in a $(k+1)$-core. Prior approaches have either ignored connectivity (which provides limited, but some insight e.g., [141]) or left the final step of finding components as a separate process. The main tool to address computing connectivity on cores, or a *core hierarchy*, has been independently proposed several times [20, 221, 80, 82] in different contexts, and concurrently developed in [169]. We introduce this index in the most basic setting, designed for k-cores on simple undirected graphs, and we call it the Shell Tree Index (ST-Index). This index supports queries to extract the cores a vertex is in along with the full core hierarchy of a graph.

Example Problem Consider the problem of managing a social network. First, given a user, we wish to recommend friends to them that are well connected *in their part* of the graph: this is a vertex and coreness query. Second, we want to detect structural changes, for example sybil attacks [63] from new fake accounts: this is a hierarchy query. Figure 4.1

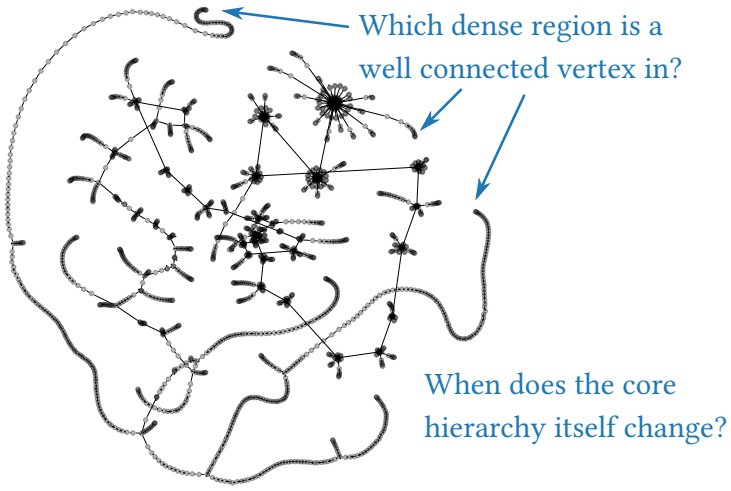


Figure 4.1: The core hierarchy for the LiveJournal social network graph. Tracking dense regions behavior over time is important for understanding structural changes, and extracting the vertices within a dense region is important for almost all known k-core applications.

shows the core hierarchy of the LiveJournal graph [267] and how far apart different dense regions are. For both query examples, we want results in tens of milliseconds to either prepare a webpage or mitigate an emerging attack.

In this example scenario, a state-of-the-art core decomposition system is put in place, which provides coreness updates quickly after graph changes. The two goals above require information about *specific cores*. If certain vertices achieve higher coreness values, this does not inform whether a new region is created. Furthermore, unless there is only one dense region, it will not enable useful recommendations. Instead, we need systems and algorithms that can quickly and effectively return *cores themselves* along with their full hierarchies.

Approach The ST-Index builds on the laminar nature of cores. For $k' < k$, every k-core is contained within some k'-core, naturally forming a tree. Each node in this tree corresponds to the *shell* of the core, that is vertices which are not in any higher core. Coupled with a reverse map, a core can be efficiently returned by traversing the subtree staying below the desired k value. The core hierarchy is the tree.

We build the tree by first identifying regions of the graph where the cores are the same,

known as subcores, and then forming a directed acyclic graph (DAG) with each subcore as a node. Starting from the highest k values, we process nodes in the DAG upwards, merging and moving them to form a tree.

The only known prior maintenance approach, operating for attributed graphs and used as part of solving community search, is given in [80]. We first port this maintenance approach to the case of k-cores on standard graphs and use that as our edge-by-edge baseline. Given an edge change, it maintains the ST-Index by either merging or splitting nodes on paths to the root. Concurrent with this work, [169] builds on [80]'s approach by batching operations on the tree.

In real-world graphs there is significant variance in the rate of change. As such, batch dynamic algorithms that can reduce the total work when operating on batches are desired [176, 60]. We provide a batch dynamic algorithm to maintain cores themselves, starting from core decompositions. We do this by maintaining the subcore DAG used during construction. After a batch of changes, we revisit each node in the DAG that was modified and re-compute any subcore changes. Any DAG changes are then pushed into the tree, temporarily turning the tree back into a DAG. We then traverse from the sink upwards, correcting the tree.

Contributions In addition to bringing the ST-Index from the community search domain into the direct, k-core domain, we prove efficiency properties on the ST-Index. Our main contributions are:

1. A subcore DAG based ST-Index construction algorithm
2. A batch dynamic algorithm to maintain ST-Index that reduces the work of edge-by-edge updates
3. An experimental evaluation on real-world graphs that show with both our edge-by-edge and batch algorithms, ST-Index is suitable for interactive use

The remainder of this chapter is structured as follows. In § 4.1 we describe the related work. In § 4.2 we formally describe our model and problem. In § 4.3 we present ST-Index. In § 4.4 we provide our algorithm to compute ST-Index from scratch. In § 4.5 we explain how to maintain ST-Index for dynamic graphs and introduce our batch algorithm. In § 4.6 we experimentally evaluate our implementations, and in § 4.7 we conclude.

4.1 Related Work

k-cores were introduced independently in [228, 183]. [183] additionally provided a peeling algorithm that uses bucketing to run in $O(n + m)$. The main strategy for computing k-cores has remained roughly the same since then: iteratively peeling the graph, or excluding vertices with too low of degrees, until all degrees are k.

For maintenance, [164] and [225] independently proposed Traversal, which limits consideration of vertices around an edge change if they provably cannot update values. [225] defines the notion of subcores and purecores, variants of which are used in all known maintenance algorithms to limit considered subgraphs. [274] proposed Order, which is the current state-of-the-art and maintains a *peeling order*, instead of coreness values directly, using an order-statistic treap and a heap. Parallel approaches have relied on identifying a set of vertices that can be independently peeled [131, 6, 121, 14]. [19, 273] provide batch algorithms that reduce work as multiple edges are processed simultaneously.

All of the above focus on computing the *coreness values* for vertices. In fact, the lack of focus on connectivity has, in some cases, resulted in later work redefining cores to not include connectivity (e.g., [180]) which limits their usefulness.

Numerous other targets, similar to cores, have been proposed [180]. [71, 275] develop weighted extensions to cores, [170] uses core concepts to reinforce connections within networks, [96] proposes notions of cores for multilayer networks, and [271] ensures vertices in core-like regions are also relatively cohesive given their neighbors. In cases where the cores are used for downstream algorithms, returning the actual (connected) vertices is identified

as crucial and algorithms are built to support such queries [171].

Community search [239, 53] is a more general problem for returning a connected set of vertices in a community based on a seed set. The community is commonly defined with a *minimum degree* measure[81]. In this case, if the query consists of a single vertex, community search can return exactly a core. For this reason, we pull from the field of community search to develop ST-Index. [20] proposed the first known shell tree index. It does not support efficient queries, as it creates additional vertices for each coreness level that must be addressed. [221] identifies the same problem that we address—cores require connectivity—and proposes a shell tree-like index with a static construction in the more general nuclei framework, but leaves out maintenance. [80] operates on attributed graphs and extends [221]'s approach and [20]'s index with incremental and decremental algorithms, but without batch algorithms. We port this approach to the problem of cores and use this as our baseline. Concurrent with this work, [169] provides a batch algorithm that is based on [80] and batches changes to the tree directly, without the use of a DAG.

4.2 Preliminaries

A graph $G = (V, E)$ is a set of vertices V and set of edges E. An edge $e \in E$ represents the connection between two distinct vertices $u, v \in V$, $e = \{u, v\}$. We denote $|V|$ by n and $|E|$ by m.

We use $\Gamma(v)$ to represent the neighboring edges of $v \in V$. The degree of $v \in V$ is $|\Gamma(v)|$. For directed graphs, Γ^{in} represents edges ending at the given vertex and Γ^{out} represents edges leaving a vertex. If the graph is ambiguous, we use Γ_G for graph G. The neighborhood of a vertex set $S \subseteq V$, $\Gamma(S)$, represents the vertices and edges connected to S, that is it is the subgraph induced by S and all neighbors of vertices in S.

4.2.1 Dynamic Graph Model

We consider graphs that are changing over time, known as dynamic graphs. An *edge change* is a tuple $\langle c, v, e \rangle$ consisting of a direction c, a vertex $v \in V$, and an edge $e \in E$. A dynamic graph is then an infinite turnstile stream of edge changes $\mathcal{S}$, where time is the position in the stream. At any point in time an undirected graph G^t can be formed by applying all edge changes until t, starting from an empty graph.

In this model, the timestamp of edges received is not preserved and not used by the algorithm. An algorithm that does take into consideration timestamps is called a *temporal* algorithm, and can either be dynamic or static.

Definition 4.1. *Let $\mathcal{A}$ be a graph algorithm with output $\mathcal{A}(G)$. Then $\mathcal{A}_\Delta$ is a dynamic graph algorithm if, for some times t and t', with $t < t'$,*

$$\mathcal{A}_\Delta \left(G^t, \mathcal{A}(G^t), \mathcal{A}^t_\Delta, \mathcal{S}[t, t'] \right) = \left\langle \mathcal{A}(G^{t'}), \mathcal{A}^{t'}_\Delta \right\rangle,$$

where $\mathcal{A}^t_\Delta$ contains algorithm state at t and $\mathcal{S}[t, t']$ are the edge changes in $\mathcal{S}$ from $t + 1$ to t'.

We call an *incremental algorithm* a dynamic graph algorithm which can only handle edge insertions and a *decremental algorithm* one which can only handle edge deletions. A *batch dynamic* algorithm can handle $t' > t + 1$. Our batch algorithm, described in Section 4.5, has an additional state bound by the size of the graph.

4.2.2 Cores

We provide a brief background on k-cores.

Definition 4.2. *Let G be a graph and $k \in \mathbb{N}$. A k-core in G is a set of vertices V' which induce a subgraph $K = (V', E')$ such that: (1) V' is maximal in G; (2) K is connected; and (3) the minimum degree is at least k,* $\min_{v \in V'} |\Gamma_K(v)| \geq k$.

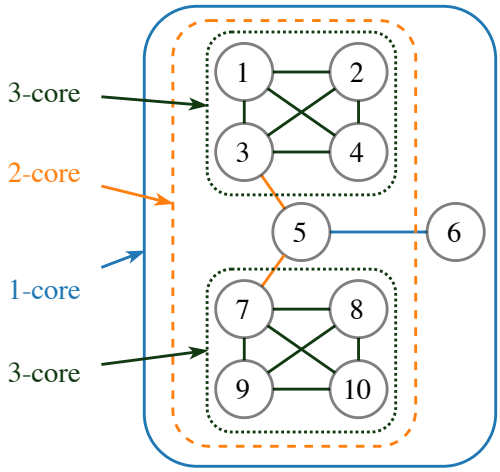


Figure 4.2: An example graph and its cores. Note that there are two separate 3-cores.

Figure 4.2 shows an example graph and its cores. There are two *separate* $k = 3$ cores, one with vertices 1 through 4 and the other with vertices 7 through 10. If all vertices with less than a degree 3 are iteratively removed, the remaining graph consists of those two separate connected components.

Definition 4.3. *Let $G = (V, E)$ be a graph and $v \in V$. The coreness of v, denoted $\kappa[v]$, is the value k such that v is in a k-core but not in a $(k+1)$-core.*

Definition 4.4. *Let $G = (V, E)$ be a graph. The k-core number of G, denoted ρ_G and shortened to ρ, is given by $\rho = \max_{v \in V} \kappa[v]$.*

4.2.3 Problem Statement

We consider the problem of efficiently supporting core and coreness queries on a dynamic graph stream. Let $k \in \mathbb{N}$ and $u \in V$.

- The coreness query $\mathcal{K}(u)$ returns $\kappa[u]$.
- The core query $\mathcal{C}(u, k)$ returns the vertices of the k-core subgraph that contains u.
- The hierarchy query $\mathcal{H}$ returns the hierarchical structure of the cores as a tree, with the root as the 0-core

Prior work in the context of cores has focused only on supporting $\mathcal{K}$ queries on dynamic graphs. Unfortunately, this prevents many of the applications of k-cores which rely on *extracting dense regions* of a graph.

4.3 Shell Tree Index

In this section we present the Shell Tree Index, ST-Index, which is able to efficiently return cores for different vertices: its runtime is asymptotically the size of the result and its space is linear in the number of vertices. This index has been independently developed several times [20, 221, 80, 82, 169] in different contexts. We present the index here for completeness. We will address how to construct the index in Section 4.4 and how to maintain it in Section 4.5.

$\mathcal{K}(u)$ queries, or *coreness* queries, can be efficiently returned using an array of size n. We therefore focus on $\mathcal{C}$ and $\mathcal{H}$ queries.

4.3.1 Naive Index

To motivate the use of ST-Index, we start by introducing a naive index which stores all of the cores for each vertex, taking $O(\rho n + n^2)$ space, using a laminar structure of cores. We then present an initial optimization that reduces the storage to $O(\rho n)$ by taking advantage of overlap in cores, but does not use the laminar structure. However, for many real-world graphs ρ is in the thousands, and so the space complexity remains too high. We then introduce a space efficient Shell Tree Index, which takes advantage of both properties of cores to achieve $O(n)$ space.

In this section we present a naive index, N-Index, which runs in optimal time but uses $O(\rho n + n^2)$ space. To efficiently support $\mathcal{C}(u, k)$ queries, we need to return the vertices in the k-core that include v. We use the following property of cores.

Lemma 4.1 ([225])**.** *Cores form a laminar family, that is every pair of cores are either disjoint or one is contained in the other.*

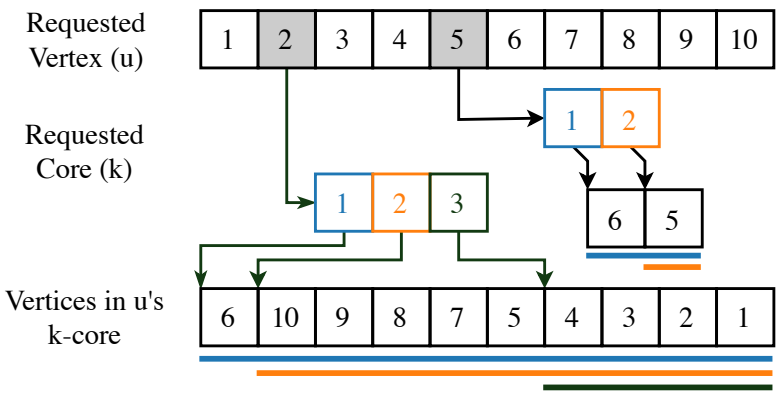


Figure 4.3: The Naive Index, N-Index, corresponding to Figure 4.2, used to return $C(u, k)$ queries. Only the second and third levels for the highlighted vertices 2 and 5 are shown, but every vertex needs its potentially unique core funnel. The colored underlines show how the cores should be read through the funnel.

Proof. We want to show that for every two cores K_1 and K_2, $K_1 \cap K_2$ is exactly one of $\emptyset$, K_1, or K_2.

Let K_1 and K_2 be two cores of G, with corresponding k values k_1 and k_2. Suppose $K_1 \cap K_2 \neq \emptyset$, implying K_1 is connected to K_2. Note that $k_1 \neq k_2$, otherwise $K_1 \cup K_2$ is a k_1-core, invalidating maximality. Let $k_1 < k_2$. Suppose $\exists v \in K_2$ such that $v \notin K_1$. v must be connected in K_2, and so there exists a path from K_1 to v with minimum degree at least k_2. Let K_1' be a subgraph that includes K_1 and the path to v. Then, K_1' is a k_1-core and larger than K_1, invalidating maximality. □

A visual representation of this index is given in Figure 4.3. We can now construct the index as follows. Build a three-level structure, with the first level corresponding to vertices and the second level corresponding to coreness values. By Lemma 4.1, we know that cores form a laminar family, and so for a given vertex the core hierarchy will consist only of smaller and smaller core regions. We call this a core funnel. We proceed by ordering vertices by their coreness from smallest to largest in the third level. Note that this funnel may be unique *for each vertex*, resulting in $O(n^2)$ storage.

The algorithm to return $C(u, k)$ in N-Index is straightforward: follow the top level pointer (u) and second level pointer (k) to get a position in the third level. Then, read out

the resulting vertices until the end of the third level list for u. The runtime is clearly the size of the output, and so it is efficient.

Lemma 4.2. *The space required for* N-Index *is* $O(\rho n + n^2)$.

Proof. The first level is size n. Every vertex may be part of a unique core funnel, and so the third level may contain n elements. The second level may contain up to ρ pointers if the vertex has maximum coreness. □

The problem with the N-Index is that the space required is too large, and importantly it ignores the relationship between vertices in the first level. We present a natural improvement next, that takes advantage of the overlap between cores for vertices.

4.3.2 Compressed Naive Index

In this section we provide a compressed naive index CN-Index, and improvement on the N-Index, that takes advantage of multiple vertices being involved in the same core. The index is modified so that what was the third level now contains cores that are shared across the whole graph. An example of this index can be found in Figure 4.4. In this case, the running time again remains efficient: for a given $C(u, k)$, the first and second level are accessed to get the core identifier. Then, the identifier is used to find and return the values exactly in the core.

Lemma 4.3. *CN-Index takes* $O(\rho n)$ *space.*

Proof. The first level contains n elements and the second level up to ρ, resulting in $O(\rho n)$. The third level has up to n elements per row, and so again takes $O(\rho n)$. □

While this is an improvement on N-Index, it does not take advantage of the laminar nature of cores. Furthermore, on many real world graphs ρ is in the thousands, making this approach untenable.

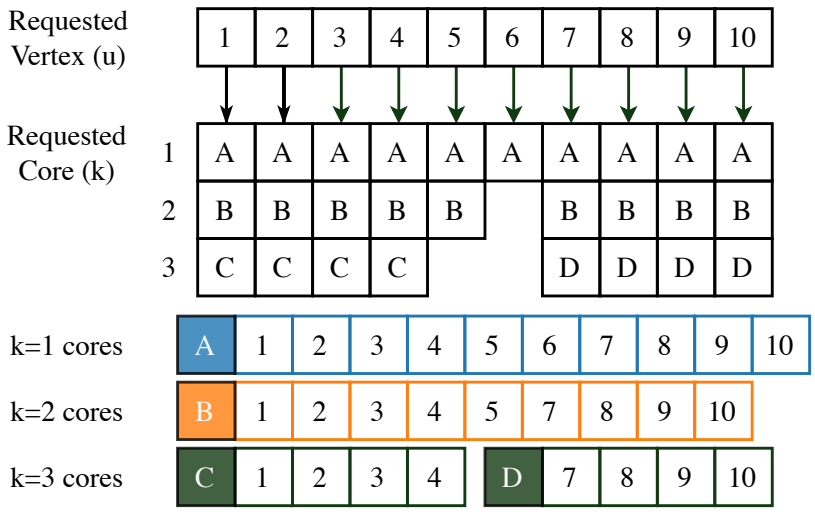


Figure 4.4: The Compressed Naive Index, CN-Index, which removes repeated vertices in the third level, but does not take advantage of the laminar nature of cores.

4.3.3 Shell Tree Index

Definition 4.5. *Let $G = (V, E)$ be a graph and $K \subseteq V$ a k-core in G for some $k \in \mathbb{N}$. Then S is a k-shell if $S = \{v \in K : \kappa[v] = k\}$.*

Note that the shell is *disconnected*, however it is a subset of a *connected* core. This means that the traditional approach of using coreness values to compute the shell does not work. We address shell computation later in Section 4.4, using *subcores*.

A *shell tree* T is at the heart of the ST-Index. We call the vertices of T *tree nodes*, to distinguish from the vertices in G. Each node has two additional pieces of data associated with it: a k value and a set of vertices (in G). T is built as follows. A root node is made with $k = 0$ and a vertex set of isolated vertices (those with $|\Gamma_G(v)| = 0$). Next, nodes are made in T for every k-shell. Its k attribute is set to k corresponding to the shell and its vertex list is set to the vertices in the k-shell. An edge is created in T by linking k-shells, following Lemma 4.1. An example shell tree is shown in Figure 4.5. The ST-Index consists of T and a map M, mapping $v \in V$ to the appropriate node in T.

Lemma 4.4. *The shell tree is a directed, rooted tree.*

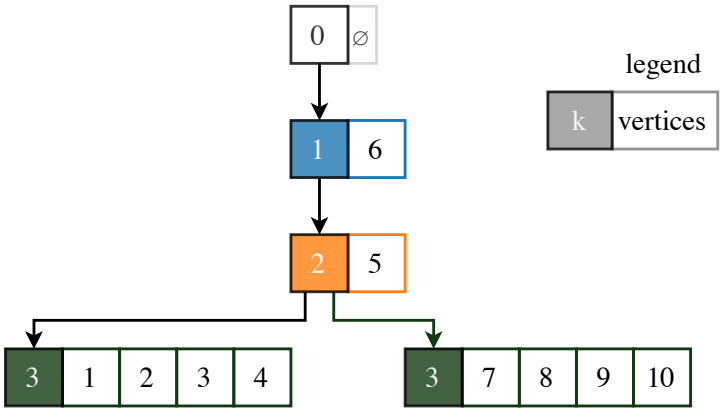


Figure 4.5: The shell tree for the graph shown in Figure 4.2. On the left side are the k-shell values, and on the right side are the contained vertices. Each directed edge indicates inclusion of the deeper cores.

Proof. Suppose a tree node u, corresponding to core K_u has two in-edges. By definition 4.5, each parent corresponds to a unique k-shell. Consider the two corresponding cores, K_1 and K_2. They both include K_u, yet are distinct, and so they have non-trivial overlap contradicting Lemma 4.1. The root is defined with $k = 0$. □

Lemma 4.5. *The out-degree of a non-root tree node with no corresponding vertices in the shell tree can be at most 1.*

Proof. Let the tree node with no corresponding vertices be at level $k > 0$ with out-degree at least 2. Then, there are two distinct *cores* at $k + 1$ (not necessarily shells), and one core at k. The two cores at $k + 1$ must be disconnected by construction.

However, because the tree node has no corresponding vertices, we know that every vertex in the k-core is also in a $(k + 1)$-core. Furthermore, the k-core is connected. Hence, it is not possible for the two cores at $k + 1$ to be disconnected. □

Lemma 4.6. *Let $G = (V, E)$ be a graph with $n = |V|$. The number of nodes in the shell tree is at most $n + 1$ and edges is at most n.*

Proof. By Lemma 4.5, each node in the tree (besides the root) must have at least one vertex. As there are at most n vertices, the size of the tree is at most $n + 1$. By Lemma 4.4, we know it is a tree, and so with at most $n + 1$ nodes it has at most n edges. □

4.3.4 Queries on ST-Index

The three queries, $\mathcal{K}(u)$, $C(u,k)$, and $\mathcal{H}$ are returned as follows:

- $\mathcal{K}(u)$ follows $M[u]$ and returns the corresponding k
- $C(u,k)$ returns vertices from a tree traversal from $M[u]$ staying above level k
- $\mathcal{H}$ returns the full tree with its nodes and attributes directly

4.3.5 Efficiency of ST-Index

We next address the efficiency of queries on ST-Index.

Theorem 4.1. *$C(u,k)$ queries on ST-Index run in $O(|C(u,k)|)$ and correctly return the k-core containing u.*

Proof. First, we show correctness. Let C^* be the core for $C(u,k)$, that is C^* is a k-core and $u \in C^*$. The traversal will cover all vertices in the subtree containing u at level k and higher. By Lemma 4.1 we know all denser cores are fully contained in the desired k-core. By Lemma 4.5, we know that any split will occur in an explicit tree node with vertices in the resulting shell. So, this split will be captured by the tree traversal. As such, all vertices in the tree nodes traversed with values k or more exactly form the k-core.

Let down represent higher k values in the tree. Next, we show efficiency. Every downward link in the subtree needs to be fully explored, and there are no nodes with overlapping vertices in the tree. Once a downward traversal occurs, there is no need to check parents. When traversing upwards, all children except the previous one will be explored downwards. In each case every node is visited exactly once and all of its associated vertices are enumerated once and are part of the returned core.

As ST-Index is a tree, whether to traverse to the parent can be decided based on whether the parents' value is lower than k. This will result in one additional operation. As such, the runtime is $O(|C(u,k)|)$ and efficient. □

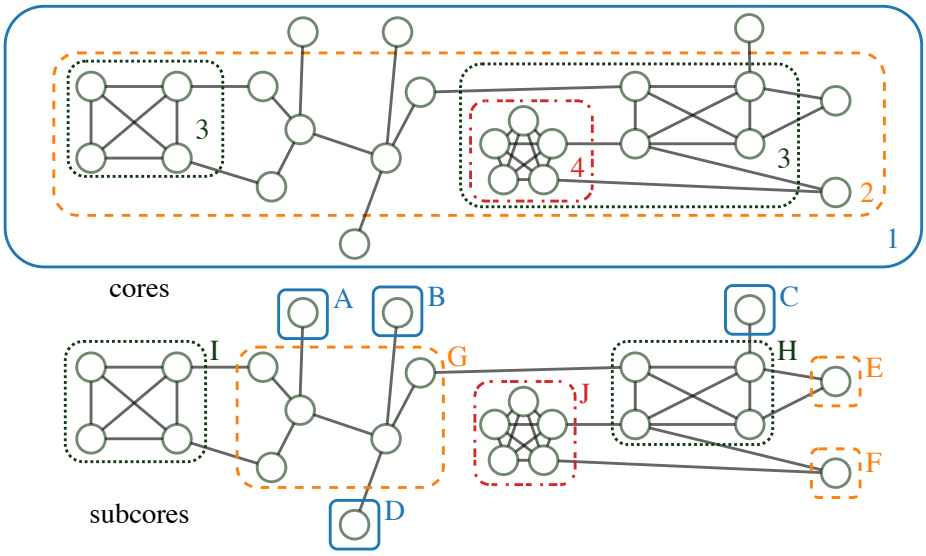


Figure 4.6: An example graph along with its cores (top) and subcores (bottom). Note that a core may consist of multiple subcores, and subcores are disjoint.

Theorem 4.2. *The* ST-Index *takes* $O(n)$ *space.*

Proof. The ST-Index consists of a map of size n between vertices and tree nodes, along with the shell tree itself. By Lemma 4.6, the tree has at most $n+1$ nodes and n tree edges. Each tree node may have vertices, but there are no redundant vertices. So, the size is $O(n+n+1+n+n) = O(n)$. □

The shell tree itself contains the hierarchy of cores and shells, and so returning ST-Index efficiently resolves $\mathcal{H}$ queries.

4.4 Computing the ST-Index

Computing (and maintaining) the ST-Index hinges on building (and maintaining) the shell tree. We propose a *subcore directed acyclic graph*, that provides the link between core decompositions and the shell tree. In this section we describe how to compute the ST-Index from scratch using the subcore DAG.

This problem is broken into three parts: computing coreness values, subcore DAG, and the shell tree.

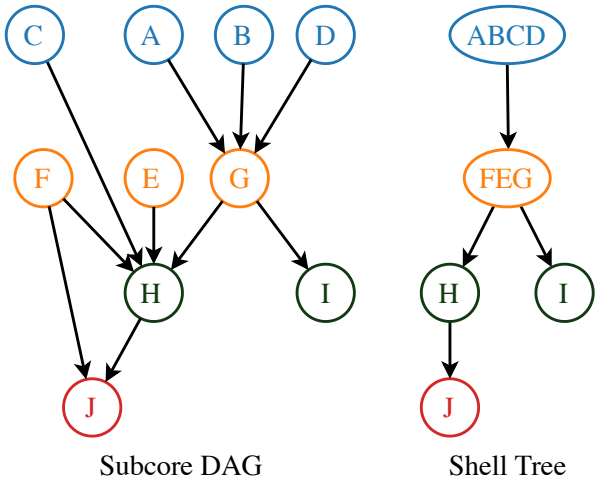


Figure 4.7: The corresponding subcore DAG and shell tree from the example graph in Figure 4.6.

4.4.1 Computing Coreness Values

Computing coreness values has been well studied on graphs [183, 58]. The most direct approach, known as peeling, starts by keeping an array of vertex degrees. It then moves up through coreness values, removing vertices with insufficient degree and recording when they are removed. This is efficient, running in $O(n + m)$, when using buckets [183]. We refer the reader to [180] for a survey.

4.4.2 Computing the Subcore DAG

Next, we introduce the *subcore directed acyclic graph (DAG)*, which is used to bridge between coreness values and cores.

Definition 4.6. *Let G be a graph. A* subcore *is a subgraph C such that (1) C is maximal (2) $\forall v \in C$, $\kappa[v] = k$ for some $k \in \mathbb{N}$ and (3) C is connected.*

Subcores were introduced in [225] to limit the region that may have coreness values change on graph changes. Figure 4.6 shows an example graph with cores and subcores.

Observation 4.1. *Subcores are disjoint, by maximality of cores and property (2), and so the number of subcores is bound by n.*

Algorithm 4.1: Building the subcore DAG.

Input: graph $G = (V, E)$, κ

$C \leftarrow \emptyset$; $D \leftarrow \emptyset$ ▷ DAG vertices and edges
$L \leftarrow [v : v \in V]$ ▷ Labels
▷ Compute the subcores
for $v \in V$ **do**
  if $L[v] \neq v$ **then** continue
  $C \leftarrow C \cup \{v\}$
  ▷ Perform a BFS that stays within κ levels from v
  $Q \leftarrow$ Queue(); Q.push(v)
  while $Q \neq \emptyset$ **do**
    $n \leftarrow Q$.pop()
    for $w \in \Gamma(n) : L[w] \neq v \wedge \kappa[w] = \kappa[v]$ **do**
      Q.push(w)
      $L[w] = v$
▷ Produce the DAG edges
for $v \in V$ **do**
  for $n \in \Gamma(v)$ *where* $L[v] \neq L[n]$ **do**
    $D \leftarrow D \cup \{\langle L[v], L[n] \rangle\}$
return DAG=(C, D)

After breaking cores up into subcores, the glue to link them back together is saved as a *subcore DAG*. The subcore DAG is built with a directed edge from every lower k subcore to a strictly higher k subcore that it is *directly connected* to. The subcore DAG from Figure 4.6 and its shell tree is shown in Figure 4.7.

Lemma 4.7. *The subcore DAG size is bound by G.*

Proof. Each vertex in the subcore DAG corresponds to a connected subgraph in the graph, and every edge in the DAG is a directed edge that results from contracting all vertices in each subcore. Contraction only removes edges and vertices, and no new edges or vertices are added. □

Observation 4.2. *The subcore DAG is not a tree. Consider a* 3*-clique and a* 4*-clique, connected via an edge, and both connected to another vertex. This forms a directed triangle in the DAG.*

The process of building the subcore DAG is shown in Algorithm 4.1. This algorithm

performs a breadth-fist search (BFS) for each vertex. The search is constrained to stay within a κ level, and DAG edges are emitted on graph edges that leave κ levels. Efficient connected components algorithms, e.g., [235], could be used instead.

Lemma 4.8. *Algorithm 4.1 runs in $O(n + m)$.*

Proof. From lines 6–11, inside the internal BFS, each vertex will be visited once. Inside, each edge will be visited once. Finally, the entire BFS will only start from unvisited vertices.

For lines 12–14, each vertex and edge will again be visited, resulting in $O(n + m)$ work. □

4.4.3 Building the Shell Tree

Given a subcore DAG and κ values, we can compute the shell tree. Our algorithm starts with the DAG and modifies it as it moves from the sinks upwards (towards lower k values), using a max-heap. Each processed vertex: 1) identifies neighbors that are at its κ level, and merges itself with them; 2) sets a single node that is an in-neighbor with the closest κ value as the tree parent; and 3) moves all other in-edges to the identified parent, ensure it becomes a tree. The details are presented in Algorithm 4.2.

Lemma 4.9. *Algorithm 4.2 correctly builds the shell tree.*

Proof. We argue that after running Algorithm 4.2, each node will exactly contain the shell. First, a node needs to contain all connected subcore DAG nodes at the given κ value. Second, it cannot have additional nodes merged with it. The vertices are processed level by level following the heap order. We argue correctness via induction on κ. At the highest κ level, by the DAG properties, we know the tree nodes connected to the sink are shells and valid. Now, consider a tree node with κ and assume nodes at $\kappa' > \kappa$ are valid. The node is formed by merging DAG nodes at the same level, which are all connected. Any connectivity that is not at level κ will be preserved by moving edges to the node's parent. By

Algorithm 4.2: Constructing the shell tree.

Input: DAG=(C, D), κ
$T = (N, E) \leftarrow$ DAG
$S \leftarrow \emptyset$
$H \leftarrow$ Heap() ▷ Empty Heap
for *sink* $s \in N$ **do**
  H.push$(\kappa[s], s)$
while $H \neq$ **do**
  $v \leftarrow H$.pop()
  if $v \in S$ **then continue**
  $S \leftarrow S \cup \{v\}$
  ▷ Merge with neighbors at same level
  while $\exists n \in \Gamma(v) : \kappa[n] = \kappa[v]$ **do**
    Merge(v, n)
    $S \leftarrow S \cup \{n\}$
  ▷ Move all remaining and new in neighbors
  $t \leftarrow \arg\max_{n \in \Gamma^{\text{in}}(v)} \kappa[n]$
  for $n \in \Gamma^{\text{in}}(v)$ **do**
    if $n \neq t$ **then** MoveEdge$(\langle n, v\rangle \rightarrow \langle n, t\rangle)$
    H.push$(\kappa[n], n)$
return T

Lemma 4.1, we know that any DAG neighbors that it is connected to will also be connected to the parent, and so the new tree node is valid. □

Lemma 4.10. *Algorithm 4.2 runs in* $O(\rho(n+m)\log n)$.

Proof. The heap processes each vertex once, and each vertex can potentially have all edges attached, resulting in $O(n+m)$ per iteration. However, edges may be carried upwards, and in the worst case all edges except one are carried upwards resulting in a factor of ρ. The log factor comes from the heap use. □

4.5 Maintaining the ST-Index

In this section, we show how to maintain the ST-Index on a graph stream. The objective is to develop a batch dynamic algorithm $\mathcal{A}_\Delta$ that will output the shell tree ST-Index, while having a small internal state $\mathcal{A}_\Delta^{\text{s}}$ and a quick runtime with low variability.

Algorithm 4.3: SingleEdge (incremental case).

Input: graph $G = (V, E)$, $e = \{u, v\}$, κ^-, κ^+, ST-Index $= (M, T)$

if $\kappa^-[u] > \kappa^-[v]$ **then** swap u, v
$K \leftarrow M[u]$ ▷ find the tree node for u
$S \leftarrow \{w \in V : \kappa^-[w] \neq \kappa^+\}$
if $M[u]$.vertices $= S$ **then**
  ▷ The entire shell moves as one subcore
  for $c \in K$.children **do**
    if c.k $= k + 1$ **then** Merge(K, c)
  K.k $\leftarrow k + 1$
  return T
▷ We need to merge or create a new sink
K.vertices $\leftarrow K$.vertices $\setminus S$
$X \leftarrow \langle K, k + 1, S\rangle$ ▷ new tree node with parent K, level $k + 1$, vertices S
for $w \in S$ **do**
  for $n \in \Gamma_{G^-}(w) \setminus S$ **do**
    if $\kappa^+[n] \geq k + 1$ **then** MergeOrConnect$(X, M[n])$
▷ Merge the path with v
$c \leftarrow M[v]$, $l \leftarrow$ SINK
while $\kappa[c] \geq \kappa^+[u]$ **do**
  $l \leftarrow c$; $c \leftarrow c$.parent
MergePaths(X, c)
return T

4.5.1 Maintaining Coreness

We refer the reader to [274, 225, 164, 91] for algorithms to maintain κ. These approaches (and similarly ST-Index) extend to trusses [50] and other nuclei [223] by use of a hypergraph [90]. For our experiments we implemented and use Order [274], the state-of-the-art decomposition maintenance algorithm.

For notational convenience, consider a time t. Let G^- denote $G^{(t)}$ and G^+ denote $G^{(t+\Delta)}$. Let κ^- denote the κ values in G^- and κ^+ denote κ values in G^+.

We take advantage of the following crucial property of coreness values on graphs: the subcore theorem.

Theorem 4.3 ([225])**.** *Let $\{u, v\}$ be an edge change. Suppose $\kappa_{G^-}[u] \leq \kappa_{G^-}[v]$. Then, only vertices in the subcore containing u may have κ values change in G^+, and they may*

Algorithm 4.4: MergePaths, which merges two paths starting from tree nodes U and V until the root.

```
   Input: ST-Index = (M, T), U, V
 1 if U = V then return
 2 if κ[U] > κ[V] then swap U, V
 3 c ← V; l ← SINK
 4 while κ[c] ≥ κ[U] do
 5 |   l ← c; c ← c.parent
 6 if κ[U] = κ[c] then
 7 |   Merge(U, c)
 8 |   return MergePaths(c, U.parent)
 9 else
10 |   MakeChild(U, c)
11 |   return MergePaths(c, U)
```

only change by 1 (increase by 1 *for insertion, decrease by* 1 *for deletion.)*

4.5.2 Single Edge Maintenance Algorithm

The main idea for maintaining the ST-Index edge-by-edge is to first break apart any core or shell that was increased and then repair the tree by merging together the paths from the endpoints. For deletions, a map is made that determines where, after a core is split, it could return to in the tree. Then, the path from the core to the root is traversed and any potential split is determined. Our algorithm shares many similarities to the community search algorithm of [80]. Our algorithm addresses cores instead of the more general community search problem on attributed graphs. Specifically, it does not need to support queries involving subsets of vertices. We refer to this approach as SingleEdge. We describe insertions in detail—deletions are similar but split nodes [80].

Let K be the tree node that has a *lower* κ *value* given an edge insertion. We first check if all of K's vertices leave. If so, we move K down and merge its children with connected subcores. Next, we iterate through the moved vertices and identify if they are connected to a shell tree node at level $k + 1$. If so, we merge those shell tree nodes together. If not, we create a new tree node for the moved vertices. Then, we walk up the tree from both

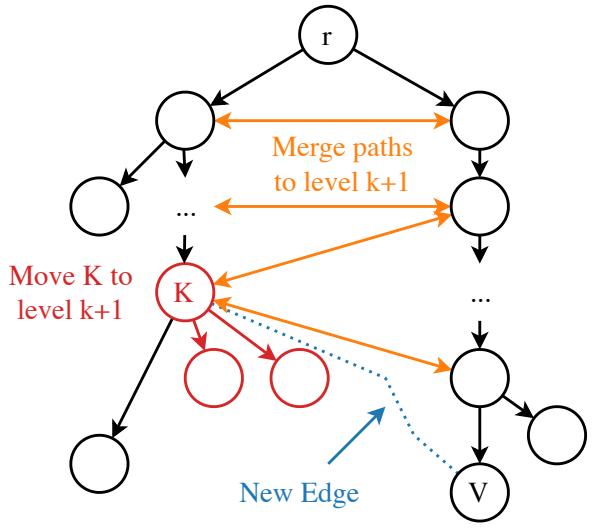


Figure 4.8: The incremental algorithm process. First, the tree node corresponding to the smaller κ level vertex, K, is processed. Next, the paths to K and to the tree node being connected are merged to level $k+1$.

endpoints and, starting at level $k+1$, begin merging all visited vertices. The algorithm is presented in Algorithm 4.3, with merge paths presented in Algorithm 4.4. A visual depiction is given in Figure 4.8.

Lemma 4.11. *The runtime for Algorithm 4.3 is $O(|\Gamma(S)|+\rho n)$, where S is the subcore that increases κ.*

Proof. In the first part, the modified subcore and all of its immediate neighbors are accessed, resulting in $O(\Gamma(S))$ work. After that, in the worst case, the height of the tree will be accessed to find the closest neighbor to merge in, resulting in $O(\rho n)$ work. □

4.5.3 Batch Maintenance

We now present our batch maintenance algorithm. First, we present the opportunity for reducing work by providing an example. In Figure 4.9, we show the graph before and after the batch.

The idea is to *keep the subcore DAG in memory* and use it to update the subcore tree. This can naturally be combined with SingleEdge to provide a hybrid approach, moving

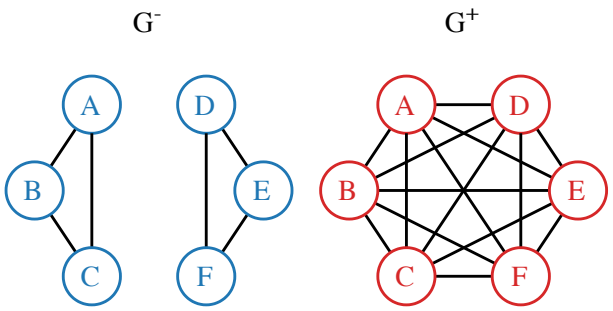


Figure 4.9: An example graph before a batch of insertions (G^-) and after (G^+). The coreness moves from $\kappa = 2$ to $\kappa = 5$ for each vertex.

between the two based on a batch size. We maintain an additional pointer between every node in the tree and every node in the subcore DAG. There are two main parts to maintaining the subcore tree in the subcore batch algorithm. First, we maintain the subcore DAG by iterating over changed vertices and recomputing any subcore changes, creating and merging subcores (locally) as appropriate. Second, we need to maintain the ST-Index given the DAG changes. To do this we begin by making all of the DAG changes propagate forward to the tree. Any deleted DAG node results in deleting the reference from the subcore tree, any newly empty tree nodes are deleted, and any new DAG nodes and their connections are added to the tree. The tree is now no longer a DAG. We then run the heap-based Algorithm 4.2 to finish turning the modified structure back into a tree. During this process we maintain the reverse vertex maps. Unlike SingleEdge, our batch approach naturally covers deletions identically to insertions and both insertions and deletions can be mixed inside of batches. This is due to handling both endpoints of an edge change, instead of only the endpoint with a lower κ value at some point in time. The approach is shown in Algorithm 4.5. Following the example in Figure 4.9, we show the saved work between SingleEdge and Batch in Figure 4.10.

Algorithm 4.5: The Batch algorithm.

Input: ST-Index = (M, T), DAG D, batch B

```
1  C ← {v : v ∈ e ∈ B}; K ← ∅
2  I ← ∅                                                    ▷ Visited set
3  for v ∈ C do
4      if v ∈ I then continue
5      I ← I ∪ {v}
6      Q ← Queue; Q.push(v)                                 ▷ Change queue
7      while Q ≠ ∅ do
8          q ← Q.pop()
9          n_d, n_T ← L[q]                                  ▷ DAG/Tree node of q
10         K ← K ∪ {n_D}
11         n'_d ← new DAG node
12         assign q to n'_d in D and M, T
13         S ← Queue; S.push(q)                             ▷ Subcore queue
14         while S ≠ ∅ do
15             n ← S.pop()
               /* Check if n is in the subcore                                  */
16             if κ⁺[n] ≠ κ⁺[q] then
                   /* If n changed, process it separately                       */
17                 if n ∉ I and κ⁻[n] ≠ κ⁺[n] then
18                     I ← I ∪ {n}
19                     Q.push(n)
20                 continue
21             if n ∉ I then
22                 I ← I ∪ {n}
23                 Q.push(n)
24             assign n to n'_d in D and M, T
25 remove newly isolated nodes in D
26 copy DAG edges from DAG nodes in K to T
27 remove newly empty tree nodes in T
28 run Algorithm 4.2
```

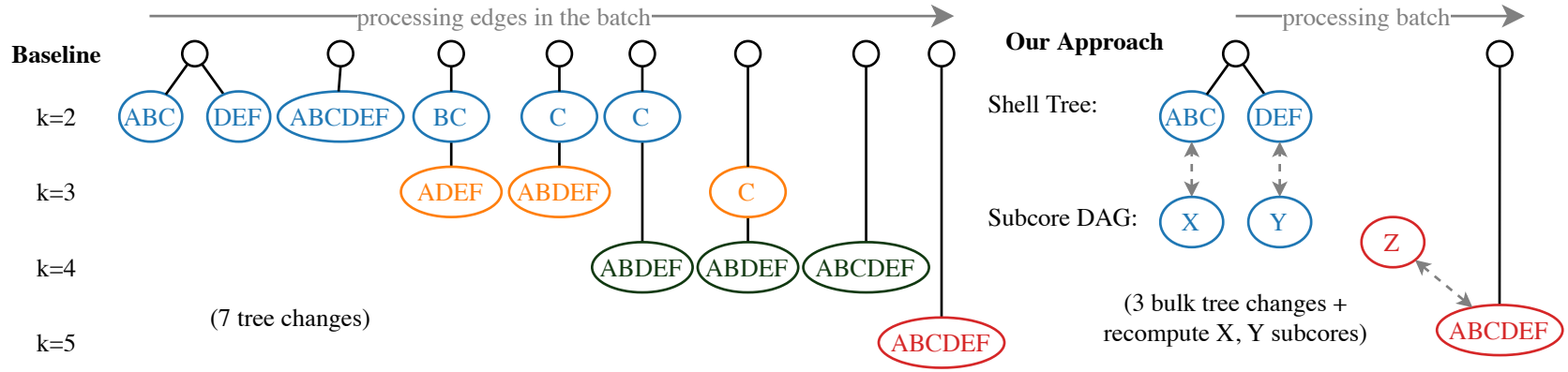


Figure 4.10: Following the example in Figure 4.9, we show the tree changes processing with SingleEdge compared with our batch approach. The cost is an increase in memory to store the subcore DAG and unnecessary work if a modified subcore does not significantly change.

Our runtime is the cost of Algorithm 4.2 plus the cost of a BFS over each modified subcore. Correctness follows from Algorithm 4.2 as we maintain the built data structures and operations. In the worst case this can be the runtime of Algorithm 4.2. However, note that the BFS on subcores is limited to modified subcores. As such, empirically we run faster than re-computing from scratch, as shown in the following Section 4.6.

4.6 Empirical Analysis

In this section we perform an experimental evaluation of our approach to demonstrate that it is able to provide core queries on rapidly changing real-world graphs.

4.6.1 Environment

We implemented our algorithm in C++ and compiled with GCC 10.2.0 at `O3`. We ran on Intel Xeon E5-2683 v4 CPUs at 2.1 GHz with 256 GB of RAM and CentOS 7. To perform coreness maintenance, we implemented Order [274]. Any coreness maintenance approach can be used in its place. We include all memory allocation costs in our runtimes. We use a hash map of vectors to store the graph, and store both in- and out-edges. We ran five trials for each experiment and show the results from all trials.

4.6.2 Baseline

As our baseline, we implemented the non-batch maintenance approach from [80], which we ported to the case of computing cores on graphs (see Section 4.5.2). We refer to this as SingleEdge. When operating on a batch, SingleEdge runs independently for each edge change. Insertions and deletions can therefore easily be mixed. We only show results with insertions as they are the harder case [80] and there are few known benchmark datasets with frequent deletions.

Table 4.1: Graphs used with n, m in millions.

Name	n, m	DAG n, m	$\lvert T \rvert$
Ar-2005 [32, 34]	22, 640	12, 47	28 K
Orkut [267]	3, 117	1, 22	254
LiveJ [267]	4, 35	2, 12	2 K
Pokec [245]	2, 22	1, 5	54
Patents [158]	4, 17	2, 4	4 K
BerkStan [161]	0.7, 7	0.2, 0.8	2 K
Google [161]	1, 4	0.4, 1.2	5 K
YouTube [267]	1, 3	1, 2.5	140

4.6.3 Datasets

The graphs that we evaluate with are benchmark graphs that are representative of real-world graphs from a variety of domains and with different properties. We downloaded them from SNAP [160] (excluding Ar-2005, downloaded from [32]). The graphs we use are given in Table 4.1. We cleaned the data by removing self loops and duplicates edges and treated graphs as undirected. We randomized the edge order, simulating a graph stream, and performed our experiments by first removing random edges and next inserting them.

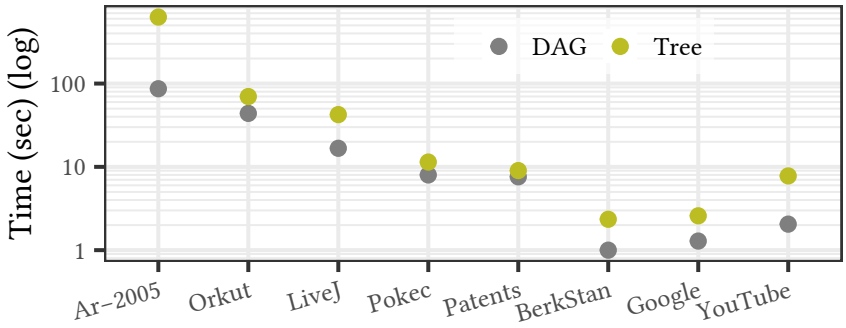


Figure 4.11: The ST-Index construction time, broken down into DAG construction and Tree construction.

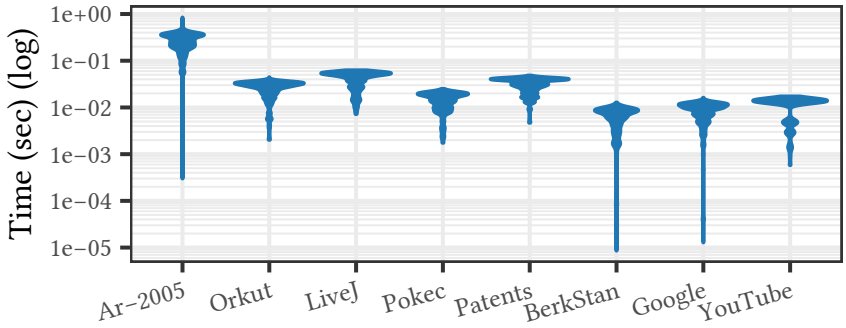


Figure 4.12: The runtime to return C queries. On all graphs, the runtimes are low enough for interactive use.

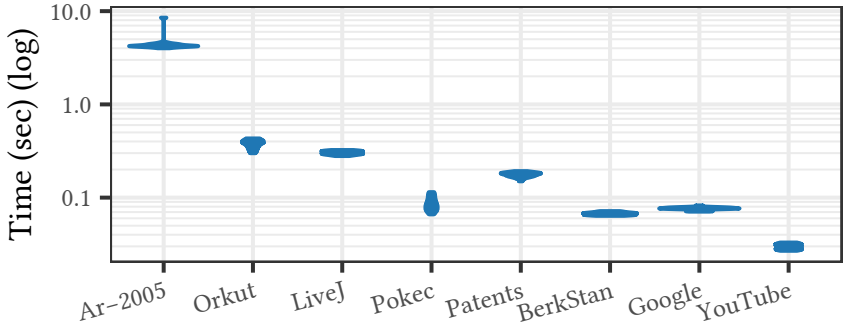


Figure 4.13: The runtime to return $\mathcal{H}$ queries.

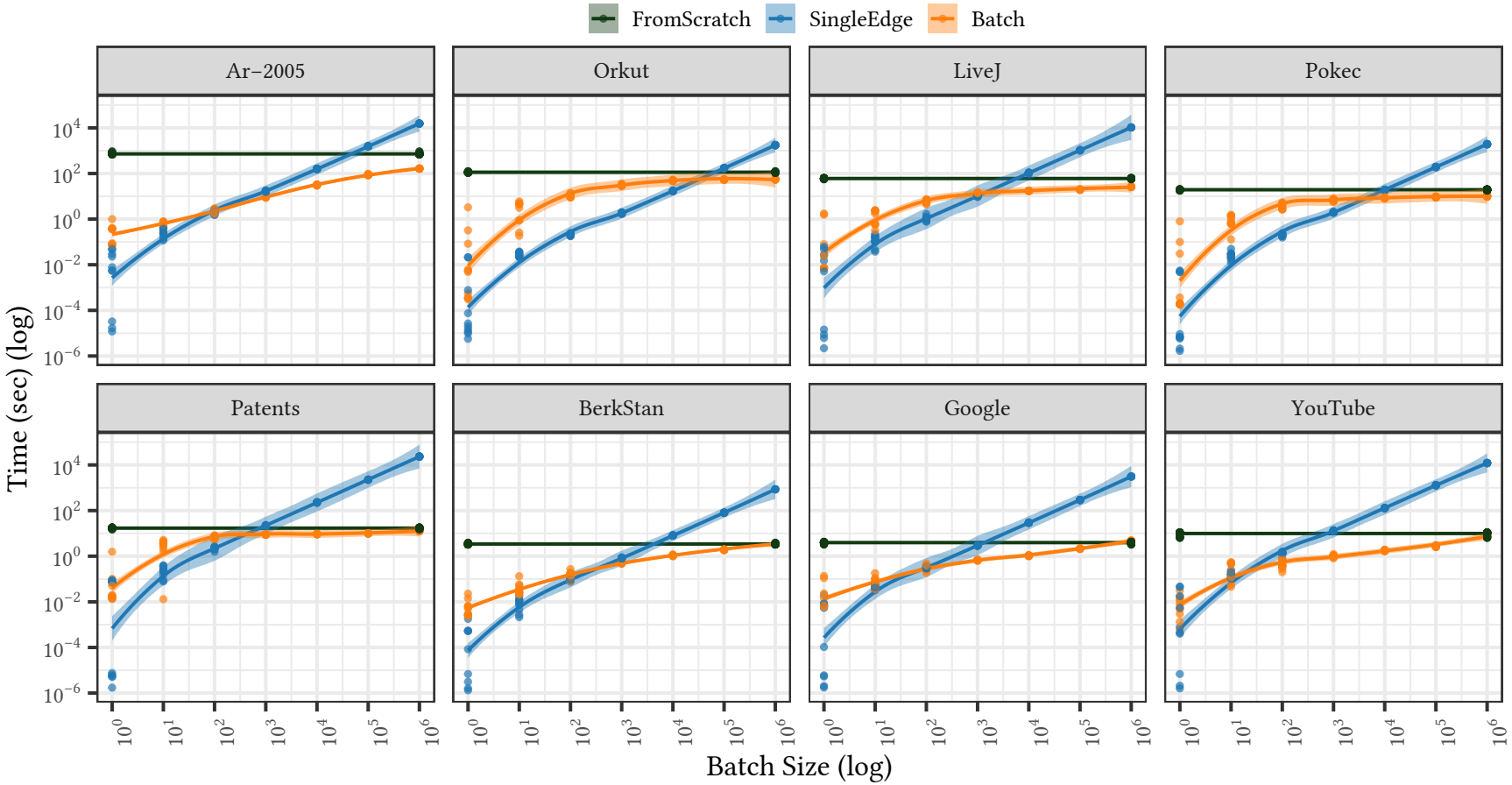


Figure 4.14: Varying the batch size and running Batch, SingleEdge, and FromScratch. The batch algorithm is orders of magnitude faster than SingleEdge for batches above 10^5 and remains below re-computing from scratch up to 10^6. The data points and LOESS smoothing lines with 95% confidence intervals are shown.

4.6.4 Experiments

Our main experimental goal is to evaluate the real-world feasibility of our approach on modern graphs and systems with highly variable and large batch sizes.

First, we show the index construction time for Batch. The results are shown in Figure 4.11. In all cases building the tree is more expensive than building the DAG. The overall runtime reinforces the need for dynamic algorithms: for large graphs, e.g., Orkut, the DAG construction takes around 90 seconds and the tree construction around 330 seconds.

Next, we want to show that ST-Index is a useful index for cores. We report the query times for C in Figure 4.12 and $\mathcal{H}$ in Figure 4.13 on ST-Index. For C, we performed queries from 1000 randomly sampled vertices with uniformly random k-values such that the vertex is in a k-core. For all graphs, all cores are returned in under one second with many in the tens of milliseconds. Given that our query is efficient the runtime largely consists of copying memory. The denser the core the faster the return tends to be, as there are fewer vertices to copy out. In many cases, the runtimes are fast enough to be used for interactive applications, e.g., in web page content. For $\mathcal{H}$, we report the time to build and return the full hierarchy, including each node at each level. This is under 10 seconds for all graphs, showing that full hierarchies can be used for interactive time applications.

Finally, we maintained cores for 100 batches of different batch sizes for each graph. The results are shown in Figure 4.14. In all cases, when batch sizes are large Batch remains below both FromScratch and SingleEdge. For a batch dynamic algorithm, we are looking for the region below re-computing from scratch and below single-edge algorithsm. In some graphs, such as Pokec and Patents, it is not a large region, however in all graphs it exists and provides significant improvements. Future work involves combining the DAG construction and maintenance with the direct tree maintenance to achieve an effective hybrid approach, achieving the lower of the all of the curves. Note that these are log-log plots, and so even for Patents our batch approach is 2× faster than re-computing from scratch at batch sizes of one million.

4.7 Summary

In this chapter, we focus on the important but overlooked problem of returning *cores*, as opposed to *coreness* values. We consider both core queries, which return a k-core, and hierarchy queries, which return the full core hierarchy. Our approach applies beyond k-cores to other arbitrary nuclei, such as trusses.

We develop algorithms around a tree-based index, the ST-Index, that is efficient and takes linear space in the number of graph vertices. We provide an algorithm to construct the ST-Index using a new approach based on a subcore DAG. We design and implement a batch maintenance algorithm for ST-Index that uses the same subcore DAG and can handle variable and high batch sizes. We show that our approach is able to run faster than edge-by-edge approaches on rapidly changing graphs and can return cores and hierarchies fast enough for interactive use.

CHAPTER 5

TEMPORAL DENSE REGIONS WITH CORE CHAINS

In this chapter, we address the increasingly important problem of identifying dense regions inside of temporal graphs that are constantly changing over time. We identify that prior approaches which find temporal regions of static vertex sets are unable to capture the dynamics of the dense regions themselves. Instead, we propose to link together dense regions in hierarchies over time, which we call core chains. These are easy to compute and define and do not require pairwise comparisons between all sets. We propose two concrete core chains, one which always includes a seed set and another which follows the majority of vertex movements. We demonstrate that these core chains are able to provide useful insight in two use cases: identifying closely related but distinct research groups as they change over time and exposing ant behavior as ants progress between different stages of life.

5.1 Introduction

Graphs have proven to be a powerful tool for studying data that has internal relationships. An increasingly important problem is to find dense regions of graphs. Such regions have proven useful for a variety of tasks, including identifying communities [148, 64], deriving news stories [13], finding link spam [103], and uncovering DNA motifs [88]. Exactly finding the densest regions of a graph is NP-hard [138], and many approximations remain hard [39, 52]. Powerful yet efficient regions to compute are k-cores [228, 183] and k-core-like structures [180], which have become standard graph analysis targets.

Many of the graphs used for analysis today are not static, and instead change over time. Graphs that preserve time information are known as *temporal graphs*. An increasingly important challenge is to develop methods to take advantage of this additional temporal information for dense region discovery [118]. Intuitively a dense region in a temporal

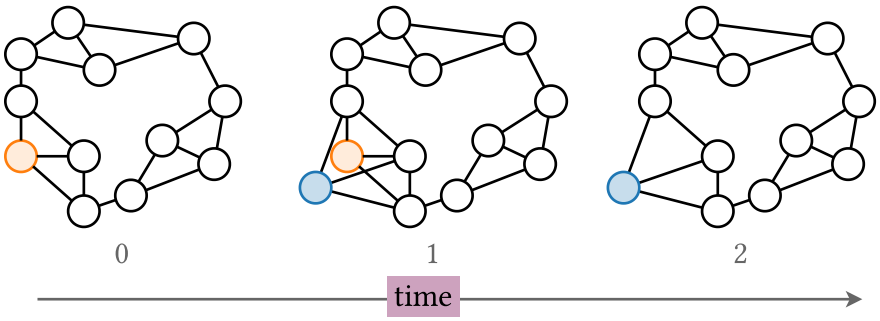


Figure 5.1: An example showing how vertex-focused approaches can fail. Intuitively, there is a dense region for three time points: the region only becomes denser at $t = 1$. All vertices are part of a 3-core at each time point. However, there is no 3-core that lasts three time points.

graph is dense over a period of time, but not necessarily over all time. There has been a recent surge of interest in finding dense regions on temporal graphs [18, 165, 123, 97, 206, 208, 167, 166, 173, 216], and there are promising applications, such as uncovering disease spread [49].

Prior work on temporal dense regions are what we call *vertex-focused*. A vertex-focused temporal region is defined by both a subgraph and a time interval, where every vertex in the subgraph matches a property for every point in time in the time interval. They are called vertex-focused because the focus of the approach is on every vertex matching some property. The main research challenges are then to jointly find a suitable subgraph and time interval. For example, a span-core [97] is a maximal set of vertices that have a degree at least k at all times within a temporal interval. Similarly an (L, K)-lasting core [123] is a maximal set of vertices with degree at least K for all points in the continuous time range L. A (θ, τ)-continual k-core [165] is a maximal set of vertices with degree at least k, which appears in every continuous time interval of length θ, with a total time interval size of at least τ. In these three examples, and more, the definition focuses on a *set of vertices* with some properties for a time interval. Unfortunately, on many temporal datasets, we show that these vertex-focused approaches do not tend to find useful dense regions.

In a temporal graph, vertices can come and go—even if a dense region remains. Fig-

ure 5.1 shows an example temporal graph with three snapshots at three time points. In this example, the orange vertex leaves and the blue vertex joins at time $t = 1$. Intuitively, the graph only *becomes denser* and so a temporal dense region should last the entire time. However, because the orange vertex leaves, no 3-core is present for the whole time and as such there is no vertex-focused temporal core, highlighting the limitations of vertex-focused approaches.

As an example, consider a department within a company that naturally forms a dense region. The company may have continual employee and management churn, yet the department persists. A vertex-focused definition would not be able to capture such a dense region. We focus on three concrete use cases where vertex-focused approaches fail: finding research groups in co-author graphs [237] that continue as students graduate, identifying good and bad users in Bitcoin trust networks [150, 149], and uncovering the maturation and progression of ants [190, 212], Overall, vertex-focused approaches are somewhat at odds with the inherently temporal nature of the dense regions themselves.

We propose a different approach that defines a new temporal, dense region, called a *core chain*, as an entity separate from its internal vertices at some point in time. At a high level, this approach *chains together* dense regions from individual snapshots of the temporal graph, and the *chain* itself is the new dense region.

Suppose we have a hierarchy of dense regions at each point in time. We call $\mathcal{D}$ the *temporal hierarchy*, and define it as a *multilayer multigraph*, where there is one layer for every time point t that contains the hierarchy of dense regions at time t. Edges between subsequent layers, e.g., from time $t-1$ to t, capture the *movement of vertices* between parts of the dense hierarchy from time $t-1$ to t. A challenge we address is to efficiently store and query $\mathcal{D}$, avoiding dense pairwise storage and computation.

As a running example, consider the dense hierarchy of the communication graph among company employees, with time discretized by day. As a day's communication differs from the prior day, the dense hierarchy at t may differ from the hierarchy at $t-1$. Each employee

present at both $t-1$ and t has a position in the corresponding dense hierarchies: for each, this position is the densest region that they are in at that point in time. A layer edge is then made in $\mathcal{D}$ for each employee, between each consecutive time points, connecting their potentially changing positions in the dense hierarchies.

We then define a core chain as a *layer path*, a set of intersecting edges in $\mathcal{D}$ that only cross layers. There are many such possible paths, but concrete path objectives and constraints can result in meaningful dense regions. We define two such constrained paths: *k-seeded core chain* and *k-majority core chain*. Both k-seeded core chains and k-majority core chains are maximal paths. Recall that inside of layers there are hierarchies of dense regions: we define these formally later, but they must be rooted trees, and the farther from the root the denser a region is. Both k-seeded core chains and k-majority core chains must be k hops away from the root. In a k-seeded core chain, the path follows edges that contain all vertices in a seed set. In a k-majority core chain, the path follows edges that contain the majority of edges from a supplied starting point.

The seed set could be, for example, a particular manager at a company. Then, the k-seeded core chain would return a chain of dense hierarchy nodes, each containing the manager (and with a density level, that is a height in the density hierarchy, of at least k). If the manager changes to another department, the k-seeded core chain would continue following the manager. For a k-majority core chain, if the dense region corresponding to a department is set as the initial dense region, then the chain would contain subsequent hierarchy nodes with the majority of staff members carrying over from the previous day. This would continue to return a dense region corresponding to the department even if the manager is replaced.

We show that k-seeded core chains are able to effectively find dense regions for research groups based on co-authorship data and k-majority core chains can uncover the behavior of ants as they mature and grow, moving from nurses to foragers.

Note that our approach applies to any dense region which is hierarchical. We evalu-

ate using nuclei [223], which generalize k-cores and provide richer and more impactful hierarchies that importantly tend to be stable.

Our computational strategy relies on maintenance algorithms. We use nuclei maintenance algorithms [90] along with k-core hierarchy maintenance algorithms [92]. Overall, we show core chains are able to find temporal dense regions when vertex-focused approaches fail. These dense regions expose behavior in the underlying data and provide greater insight that is otherwise lost or unavailable.

Contributions. Our main contributions are as follows:

- We define core chains, which are inherently temporal dense regions that are robust against vertex and edge changes
- We provide an effective data structures and algorithm to build and query $\mathcal{D}$
- We propose two concrete core chains, k-seeded core chain and k-majority core chain and provide algorithms to compute them
- We evaluate the effectiveness of k-seeded core chain and k-majority core chain using two case studies

The remainder of this chapter is structured as follows. In § 5.2 we describe preliminaries and related work. In § 5.3 we define core chains and the two concrete instantiations, k-seeded core chains and k-majority core chains. In § 5.4 we propose algorithms and data structures to build and query $\mathcal{D}$. In § 5.5 we evaluate k-seeded core chain and k-majority core chain with three use cases and in § 5.6 we conclude.

5.2 Preliminaries and Related Work

5.2.1 Preliminaries

Graphs A graph $G = (V, E)$ is a set of vertices V and edges E. An edge $e = \{u, v\} \in E$ indicates a connection between vertices $u, v \in V$. The neighbors of a vertex v in G are

denoted $\Gamma_G(v) = \{u \in V : \{u, v\} \in E\}$. The degree of u is given by $\deg_G(u) = |\Gamma_G(u)|$. G can be dropped, if there is no ambiguity. An l-clique is a fully connected graph on l vertices. A path is a sequence of connected edges that visit each vertex at most once. A cycle is a sequence of connected edges that start and end at the same vertex. A tree $T = (V, E)$ is a connected graph with no cycles. A hypergraph H is a generalization of graphs, $H = (V, E)$ where V is again a vertex set but each edge $e \in E$ is a subset of vertices, i.e., $e \subseteq V$. A multigraph follows the graph definition, but edges are a multiset instead of a set (and so identical edges may be present). A multilayer multigraph is conceptually a collection of graphs, called layers, where each layer typically shares a vertex set with other layers and has edges that remain within the layer. Additional edges cross between layers. Formally, a k-layered multigraph $\mathcal{G} = ([G_1, \ldots, G_k], \mathcal{E})$, where G_t is a graph and $\mathcal{E} = \{(l_1, l_2, \{u, v\}), \ldots\}$ is a multiset representing edges crossing between layers, with $l_1, l_2 \in \{1, \ldots, k\}$ and $u, v \in \bigcup_{i=0}^{k} V_i$. A layer path in a multilayer multigraph is a sequence of edges in $\mathcal{E}$ that connect distinct graphs G_t.

Given a discrete time domain T, a temporal graph $\mathcal{G} = [G_t]_{t \in T}$ is a collection of graphs for each time point, where vertices and edges may change between time points. This can be represented as a stream of edge changes with timestamps, $\mathcal{G} = (\{u, v, t, \pm\}, \ldots)$, where $u, v \in V$, $t \in T$, and $\pm$ indicates whether the edge is created (+) or deleted (−) to arrive at G_t from the previous time, starting from $G_0 = (\emptyset, \emptyset)$.

In many cases, temporal data only directly indicates *interactions*, and to create a temporal graph from these datasets it is helpful to understand the natural lifespan of the interaction. In these cases insertions can be created when the interaction begins, and a corresponding edge deletion is created at a time when the interaction no longer has meaning. This process is known as *ageing out* edges. For example, in the case of ants, any interaction between ants may naturally age out after one day [190].

A *maintenance algorithm* operates on a stream of graph edge changes, but it does not take into consideration time. Given a prior state and a batch Δ of edge changes, a mainte-

nance algorithm $\mathcal{A}_\Delta$ will return the same output as a static algorithm that runs on the entire known graph. Maintenance algorithms are useful to improve performance for keeping an up-to-date algorithm output as the graph changes, even if they do not provide asymptotic runtime improvements over re-computing [77].

C-trees C-trees were introduced to quickly handle updates to dynamic graph data structures [59]. A C-tree is a method of storing an ordered list of elements. At a high level, there is some probability that an element in the list will become *promoted* and otherwise, the element will be attached to the closest promoted element in the ordered list before it. The technique used to randomly choose an element to promote is to hash the value, with a randomly random hash function, and see if the resulting hash is zero. Note that the promoted elements are still ordered, but there are a lot fewer of them than elements in the full list. These promoted nodes are then placed in a balanced binary tree.

Let $S = (s_1, \ldots, s_n)$ be an ordered list, h be a uniformly random hash function, and $b \in \mathbb{N}$ a constant. A C-tree will promote some of the elements to tree nodes, and the remaining elements will be attached to the tree node before them, defining the node's value. An element $s \in S$ is promoted if $h(s) \bmod b = 0$. The value associated with s is a variable sized ordered list $(s_j, \ldots, s_k)$ where $s_j, \ldots, s_j$ are greater than s and less than s', where $h(s') \bmod b = 0$ is the *next* promoted element. Finally, any elements prior to the first promoted element are considered the *prefix* and stored as a separate block. The promoted elements are stored in a balanced binary tree.

When the ordered list that a C-tree represents is updated, the binary tree will need a path to the root to be updated, but the associated values will only change if those elements change.

Cores and Nuclei Cores [228, 183] are connected regions of a graph that have a *self sustaining* minimum degree. If the remainder of the graph is removed, the core retains its minimum degree. Nuclei [223] are generalizations of cores that are parameterized by

two integers, r and s. We explain nuclei through (r, s)-hypergraphs. Formally k-core, (r, s)-hypergraph and k-(r, s) nuclei can be defined as follows:

Definition 5.1 (k-core [228, 183])**.** *Let $k \in \mathbb{N}$ and let G be a graph. A k-core is a maximal connected subgraph induced by $K \subseteq V$ such that $\forall v \in K$, $\deg_G(v) \geq k$. The largest k value such that a vertex v is part of a k-core but not a $(k+1)$-core is denoted $\kappa[v]$.*

Definition 5.2 ((r, s)-hypergraph [90])**.** *Let $k, r, s \in \mathbb{N}$ with $r < s$ and let G be a graph. The (r, s)-hypergraph has a vertex for each r-cliques in G and a hyperedge for each s-clique, where each hyperedge connects all $\binom{s}{r}$ r-cliques in the corresponding s-clique.*

Definition 5.3 (k-(r, s) nuclei [223])**.** *Let $k, r, s \in \mathbb{N}$ with $r < s$ and let G be a graph. A k-(r, s) nucleus is a k-core in the (r, s)-hypergraph. Note that hyperedges, similar to edges, cannot be broken: either the entire edge is in a subgraph or none of it is. The largest k value for an r-clique such that the r-clique is part of a k-nucleus but not a $(k+1)$-nucleus is denoted $\kappa[r]$.*

There are several maintenance algorithms for k-cores [180]. The Order algorithm [274] is the current state-of-the-art decomposition maintenance algorithm. However, this only maintains the density levels for each vertex. Efficient algorithms have been developed to then maintain the full hierarchy [92]. Furthermore, arbitrary (r, s) nuclei maintenance is possible by first maintaining the (r, s)-hypergraph and then maintaining hypergraph k-cores [90].

5.2.2 Related Work

Finding dense regions of temporal graphs is an increasingly important topic, with a rapid increase in work over the last several years. Prior work can be categorized based on how density is defined. The definition we consider views density as the ratio of edges in a subgraph to the number of possible edges, where the densest region is then a clique. Using this definition, Wu et al. and Bai et al. [264, 18] project the temporal graph to a weighted

graph by adding 1 to every edge weight for every snapshot the edge is in, and then find weighted cores. Aggarwal et al. [4] builds a probabilistic pattern to match high density regions defined as pseudo-cliques. Yang et al. [268] looks at quasi-cliques that are present within a time range. There are many approaches that consider cores in any sufficiently long interval, with different properties such as being diverse, periodicially occuring, or lasting a maximal amount of time [54, 95, 97, 206, 208, 163, 167, 123, 165]. Lin et al. [166] considers quasi-cliques that need to be present on average within a time range. Yu et al. [269] finds historical k-cores and develops an index to quickly query them, but does not explicitly find temporal cores.

All of the above approaches are *vertex-focused*. A vertex-focused temporal region is a subgraph and a time interval, and the subgraph needs to exist and have some property for the whole time interval. A vertex-focused temporal core, for example, is a k-core that must exist for the whole time range. They will fail to uncover a dense region when the region itself is undergoing change: instead, they find *static* regions that exist for some period of time.

There are several other approaches that consider density as the average degree, sum of weights, or as a cohesiveness measure [207, 172, 173, 130, 229, 216, 31, 177, 48]. Other approaches find periodic subgraphs and motifs, but do not require them to be dense [153, 24].

A related problem to finding dense regions is to perform *community detection* or *clustering*, which identifies potentially overlapping communities in datasets [86, 118, 108]. This differs in focus from finding dense regions by focusing more on individual *vertex membership* in a community, instead of considering the *graph structure*. While some community detection algorithms are hierarchical, the trade-off from the shifted focus can result in highly variable swings during only minor—or even no—graph changes [94]. There are many similar ideas for core chains and dynamic community detection tracking. In both approaches, hierarchies or communities exist for each time point, and the problem is in

how to connect them to identify the temporal structure. Core chains do not show promise when the hierarchy radically changes from time point to time point, and similarly temporal community connection approaches [108] do not apply well to stable hierarchies, such as those from nuclei.

5.3 Core Chain Definition

In this section we first define core chains and then propose two concrete core chains that identify dense, temporal regions in graphs.

We build core chains off of *density hierarchies*. We build up to the definition of a density hierarchy in this section, but the intuition is that some regions (subgraphs) are denser than others. By the nature of density, if a region is very dense, there is likely a larger and less dense region that the dense region is also part of. At the extreme, all dense regions are part of the entire graph, which may be very sparse. This implies a hierarchy can be built, where the regions connect in a rooted tree where the root ultimately contains the entire graph.

While our definition and approach is for density hierarchies in general, we provide examples using k-cores (Definition 5.1) for clarity. Note that any vertex in a k-core with $k > 1$ is also in a $(k-1)$-core: if all vertices in the subgraph have a degree at least k, then they all also have degree at least $k-1$. As such, the k-cores of a graph can be represented by a *tree*, where the root is a special node representing the whole graph, and all other nodes represent cores in the graph. The distance from the root is the level k of the k-core, and the immediate neighbors of the root are the 1-cores (connected components) of the graph. As nodes in the tree are farther from the root, they have a *higher* k value.

We call such a tree a *density hierarchy tree*.

Definition 5.4. *Let $T = (N, L)$ be a rooted tree with root r, tree nodes N, and tree edges L, where $\{n, m\} \in L$ with $n, m \in N$. Then T is a density hierarchy tree, where each node $n \in N$ represents a dense region. The root $r \in N$ is a special node that represents the entire graph. The distance from r to node $n \in N$ represents the density level of n.*

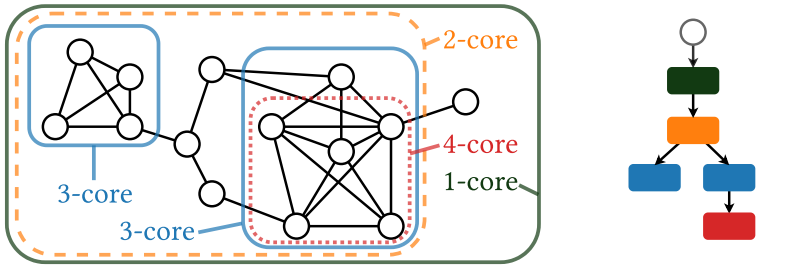


Figure 5.2: A graph, its cores, and its density hierarchy.

Consider the example in Figure 5.2. On the left we show a graph along with its k-cores. There is a 1-core that represents the connected component. In orange there is a 2-core, in blue there are two different 3-cores, and in red there is the densest region, a 4-core. The density levels are hierarchy: everything in the 4-core is also in one of the 3-core, which are in the 2-cores and ultimately everything is in the 1-core. The corresponding *density hierarchy tree* is shown on the left. The small circle corresponds to the root, the dark green the one core, the orange the two cores, and so on. An edge in the hierarchy represents that the denser region, which is farther from the root, is *contained inside* of the closer region.

Next, we need to capture how the *vertices* in the graph are mapped into the *tree nodes*. Due to the hierarchical nature of the density hierarchy tree, we know that if a vertex is in a dense region, then it is also in all of the other dense regions along a path to the root of the tree. Furthermore, a node cannot be in two separate dense regions—otherwise it would not be a hierarchy. This means that, for each vertex, it is sufficient to map it to the just the densest node in the density hierarchy: all of the other less denser regions it is part of can be explored by walking from the node towards the root.

Definition 5.5. *Given a density hierarchy tree $T = (N, L)$ with root r, and a graph $G = (V, E)$, a* density mapping *$M : V \to N$ maps every vertex to the farthest tree node away from the root corresponding to a dense region the vertex is in.*

Both the tree and the mapping together are called a *density hierarchy*, which is what core chains are built on.

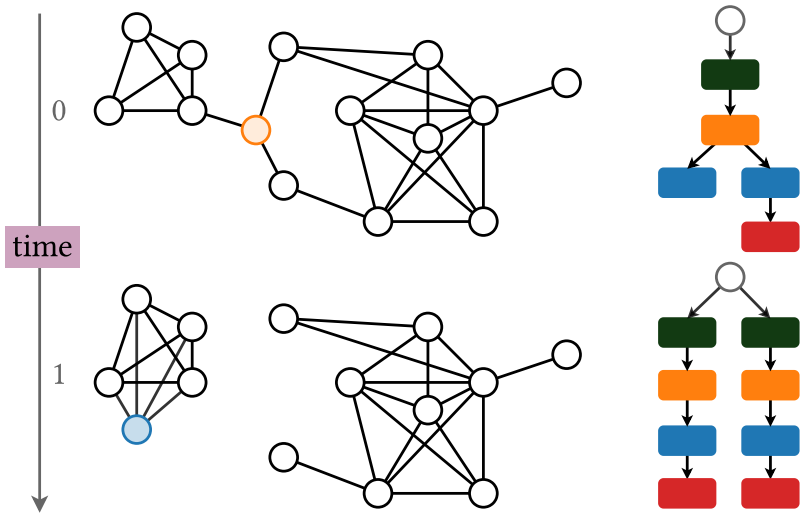


Figure 5.3: An example temporal graph with two time points, each shown by the corresponding graph snapshot on the right side. Between the first and second time points, the orange node (and associated edges) are deleted, and the blue node (and its edges) are added. The density hierarchies for each time point is shown on the left side.

Definition 5.6. *For a graph* $G = (V, E)$*, a* density hierarchy $D = (T, M)$ *consists of a rooted density hierarchy tree* $T = (N, L)$ *with root* r *and a density mapping* $M : V \to N$.

In a temporal graph, the density hierarchy may differ between points in time. We denote the hierarchy at time t as $D_t = (T_t, M_t)$. The maximum depth of all hierarchies is denoted ρ.

An example temporal graph and its density hierarchies at two points in time, derived from k-cores, are shown in Figure 5.3.

Consider the *vertex movement* between dense regions at points in time. Note that this is separate from the density mapping changes—and it is separate from any temporal graph changes. This *movement* indicates what dense regions a vertex is in after the time change. For example, if a vertex starts in a 2-core, note that it is also in the 1-core and in the root (the 0-core). If the density drops from a 2-core to a 1-core, then its *vertex movement* would be the set of all possible changes: $\{(0,0), (1,0), (2,0), (0,1), (1,1), (2,1)\}$, where (a, b) indicates it is in dense region a at the first timestep and it *moves to* b at the second.

By definition, we know that M_t will map a vertex into the densest region that the vertex

appears in at time t, and due to the hierarchical nature of D_t it exists in all less dense regions to the root. To build all vertex movements, we create an entry for every pair of tree nodes that a vertex is in between two consecutive time points.

Definition 5.7. *Let T be a time range, $\mathcal{G} = [G_t = (V_t, E_t)]_{t\in T}$ be a temporal graph, and $D_t = (T_t, M_t)$ be the density hierarchy for time $t \in T$. Let $P_t(v)$ be the vertices along the path in T_t from $M_t(v)$ to the root of T_t. Then, the* vertex movement *for v from t to $t+1$ is given by $M_t^{t+1}(v) = \{\{a, b\} : a \in P_t(v), b \in P_{t+1}(v)\}$.*

We can now define the main datastructure used in core chains.

Definition 5.8. *Let T be a time range, $\mathcal{G} = [G_t = (V_t, E_t)]_{t\in T}$ be a temporal graph, and $D_t = (T_t, M_t)$ be the density hierarchy for time $t \in T$. Then the* temporal hierarchy *is a multilayer multigraph $\mathcal{D} = ([G_t]_{t\in T}, \mathcal{E})$, where*

$$\mathcal{E} = \bigcup_{[t,t+1]\in T} (t, t+1) \times \bigcup_{v\in V_t\cap V_{t+1}} M_t^{t+1}(v).$$

Lemma 5.1. *Let T be a time range, $\mathcal{G} = [G_t = (V_t, E_t)]_{t\in T}$ be a temporal graph, and $\mathcal{D} = ([T_t = (N_t, L_t)]_{t\in T}, \mathcal{E})$ be a temporal hierarchy for $\mathcal{G}$. Let $n = \max_{t\in T} |V_t|$. $\mathcal{D}$ has at most $O(|T| \max_{t\in T}(|N_t|)^2 n)$ edges.*

Proof. Every vertex may contribute its edges at each time point. In the worst case, the vertex may be part of every dense region, and hence there are $\max_{t\in T}(|N_t|)^2$ edges per vertex. This bound is tight: consider a clique that does not change and the density hierarchy from k-cores on it. □

The temporal hierarchy captures both the density hierarchies at all times and the vertex movement between different points in the density hierarchies. Figure 5.4 shows an example temporal hierarchy.

Definition 5.9. *A* core chain *is a layer path in $\mathcal{D}$.*

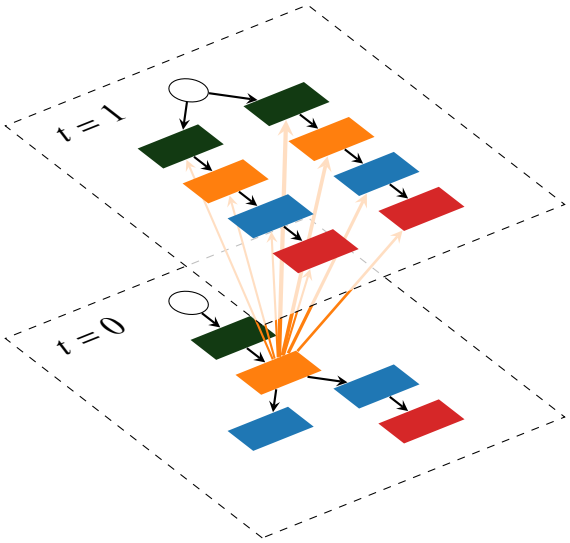


Figure 5.4: The temporal hierarchy corresponding to the graph in Figure 5.3, showing each possible dense region a vertex can move between from $t = 0$ to $t = 1$ after starting in the $k = 2$ core. Edge size indicates the number of multiedges. Note this temporal hierarchy, with numerous edges, does not need to be explicitly stored or built.

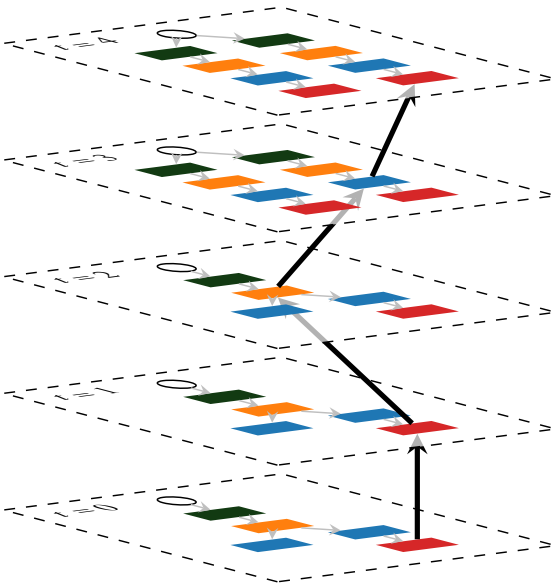


Figure 5.5: A core chain through a temporal hierarchy with 5 timesteps.

A core chain is shown in Figure 5.5. The core chain is a layer path through $\mathcal{D}$. The vertices that are part of the core chain change over time, as the graph changes temporally. The dense object itself is the path.

There are many possible core chains, and we want to focus on those that uncover useful and important dense regions over time. We define two concrete core chains that return such temporal dense regions.

5.3.1 k-Seeded Core Chains

A k-seeded core chain is used to return a dense region that always contains a *seed set* of chosen vertices.

Definition 5.10. *Let T be a time range, $\mathcal{G} = [G_t = (V_t, E_t)]_{t \in T}$, $S \subseteq \bigcup_{t \in T} V_t$ be a set of* seed vertices*, and $\mathcal{D}$ be the temporal hierarchy of $\mathcal{G}$. A k-seeded core chain is a core chain in $\mathcal{D}$ that is* maximal *with respect to both the path length and the sum of k values in the chain and any tree node n in the layer path for the core chain obeys the constraints: (1) each vertex of S maps to n and (2) the depth of n is at least k.*

Note that there may be more than one k-seeded core chain in a given temporal graph for a given seed set S, as S may come and go in dense regions in different periods of time.

This core chain is useful if a group of vertices is known a-priori to be of interest. It not only determines when those vertices are densely connected, but also any other vertices as time goes on that are densely related. As an example use, we later explore k-seeded core chain for research groups in co-author graphs where students change over time but the group, which contains a lead professor as the seed set, remains a dense region.

5.3.2 k-Majority Core Chains

Instead of following a set of seed vertices, a k-majority core chain follows the *majority* of vertices from a given starting point.

Definition 5.11. *Let $\mathcal{G}$ be a temporal graph, $\mathcal{D}$ be the temporal hierarchy of $\mathcal{G}$, and n be a vertex in $\mathcal{D}$. The k-majority core chain is a core chain in $\mathcal{D}$ that is maximal with respect to the path length and any tree node in the core chain layer path obeys the constraints: (1) the edge leaving n in the core chain is the edge with the largest number of multiedges and (2) the depth of n is at least k.*

A full decomposition provides the k-majority core chain for each node in the temporal hierarchy.

This core chain is useful to identify a dense region that remains active over time, independent of individual group membership. Later, we show that k-majority core chain is able to find two communities of ants, one in which members stay close to the queen ant (called nurses) and another in which members travel outside to find food (called foragers). These communities persist, even though the ants themselves tend to move from being nurses to being foragers over their lifespan [212].

Consider the example in Figure 5.4. If we want to compute the 1-majority core chain starting from the 2-core at $t = 0$, then the chain would progress to the $k = 1$ core on the right hand side, as this has the majority of vertices (9 vertices). If we find the 2-majority core chain, then the chain would proceed to the $k = 2$ core on the right hand side (with 7 vertices).

5.4 Computing Core Chains

In this section we describe how to compute core chains. Computation is broken into two parts. First, we need to construct a datastructure to represent $\mathcal{D}$. Second, we need to find an appropriate layer path in $\mathcal{D}$.

At a high level, we build $\mathcal{D}$ by focusing on *shell movement* instead of the full, vertex movement (from Definition 5.7). This results in a much smaller multilayered graph that is easier to maintain, but still supports fast enough queries to return paths. To compute paths in $\mathcal{D}$, the layer edges are enumerated by first walking up and down the tree, ensuring the

depth is at least k, and collecting layer edges. Those layer edges are then traversed and corresponding walking in the tree occurs in the next layer.

5.4.1 Shell Temporal Hierarchy

First, we introduce *shells*.

Definition 5.12. *Let $G = (V, E)$ be a graph and $D = (T, M)$ be the density hierarchy for G. Consider some $n \in T$, at level k. The* shell *is given by $S(n) = \{v \in V : M(v) = n\}$.*

That is, the shell consists of vertices that are in a level k, but not in any level $(k + 1)$. The shell of the root contains isolated vertices.

We introduce $\mathsf{shell}\text{–}\mathcal{D}$ as a reduced form of $\mathcal{D}$ that only contains multiedges with destinations that are in a *shell*.

Definition 5.13. *Let T be a time range, $\mathcal{G} = [G_t = (V_t, E_t)]_{t\in T}$ be a temporal graph, and $D_t = (T_t = (N_t, L_t), M_t)$ be the density hierarchy for time $t \in T$. Then the* shell temporal hierarchy *is a multilayer multigraph $\mathsf{shell}\text{–}\mathcal{D} = ([G_t]_{t\in T}, \mathcal{E})$, where*

$$\mathcal{E} = \bigcup_{[t,t+1]\in T} (t, t+1)\times \bigcup_{v\in V_t\cap V_{t+1}} \left\{ \begin{array}{r} \{a, b\} \in M_t^{t+1}(v) : \exists m \in L_t, a \in S(m) \\ \text{and } \exists n \in L_{t+1}, b \in S(n) \end{array} \right\}.$$

Using shells, we can capture the movement of vertices through dense regions in a much more concise manner than saving the entire temporal hierarchy $\mathcal{D}$.

In Figure 5.6 we show $\mathsf{shell}\text{–}\mathcal{D}$ containing all edges between $t = 0$ and $t = 1$. Due to the nature of the shells, it is possible to reconstruct $\mathcal{D}$ from $\mathsf{shell}\text{–}\mathcal{D}$: starting from every node in the tree, duplicate the multiedges up to the root for every vertex in the shell.

Storage Complexity and Compression By using a shell structure we are able to significantly reduce the storage requirements from storing the entire $\mathcal{D}$.

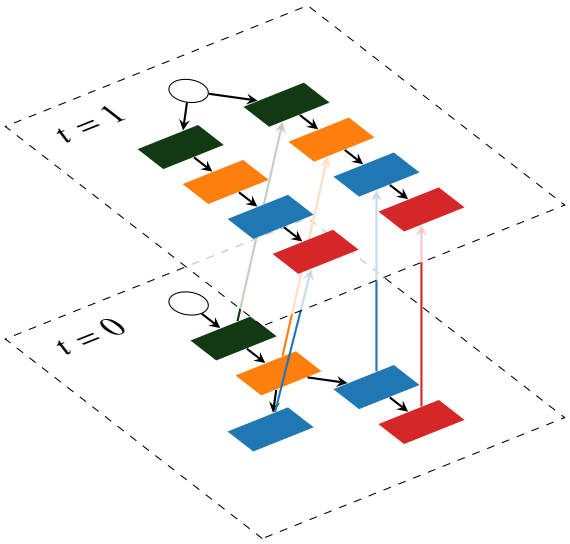


Figure 5.6: A shell temporal hierarchy corresponding to the temporal hierarchy in Figure 5.4. By moving through the tree at each layer, the vertex movements can be reconstructed.

Lemma 5.2. *Let T be a time range, $\mathcal{G} = [G_t = (V_t, E_t)]_{t \in T}$ be a temporal graph, and* shell$-\mathcal{D}$ *be a shell density hierarchy for $\mathcal{G}$. Let $n = \min_{t \in T} |V_t|$. Then* shell$-\mathcal{D}$ *has a storage complexity of $O(|T|\, n)$.*

Proof. At each time point, the hierarchy may arbitrarily change. However, the number of edges is bound by the number of vertices in each layer. Furthermore, the density hierarchy is bound by the number of vertices. Hence, the space is bound by $O(|T|\,(n+n))$. □

Furthermore, we can apply a compression technique using C-trees. We store each *layer* in shell$-\mathcal{D}$ as a separate *root* of a C-tree. In effect, when the temporal structure is modified, a new root is created. We store $|T|$ roots, and $|T|$ paths to modified structures. However, we do not have to replace the entire shell$-\mathcal{D}$ structure at each time point.

This has a considerable cost savings when there are many time points that do not significantly change the hierarchy. In these cases, a common storage cost for a time points will be $O(\log \rho)$, as the density hierarchy between layers only needs to be modified slightly.

Algorithm 5.1: Computing shell–$\mathcal{D}$ using purely-functional data structures and hierarchy maintenance algorithms

Input: temporal graph $\mathcal{G} = ([G_t]_{t \in T})$
Output: shell–$\mathcal{D}$

$D \leftarrow \mathcal{A}(G_{t_0})$ ▷ compute the first density hierarchy
$L \leftarrow [D_{t_0}]$ ▷ layers begins with the first hierarchy
$\mathcal{E} \leftarrow \emptyset$ ▷ no layer edges exist initially
for *pair* $(t, t+1) \in T$, *ordered* **do**
  $\Delta_{t+1} \leftarrow G_{t+1} - G_t$ ▷ compute batch changes
  $M_\Delta \leftarrow \mathcal{A}_\Delta(D, \Delta_{t+1})$ ▷ maintain D to get D_{t+1}, return M changes
  for $v \in M_\Delta$ **do**
    $\mathcal{E} \leftarrow \mathcal{E} \bigcup (t, t+1, \{M_t(v), M_{t+1}(v)\})$
  $L \leftarrow L \bigcup D_{t+1}$
return shell–$\mathcal{D} = (L^0, \mathcal{E})$ ▷ L^0 returns all density trees of L

5.4.2 Computing shell–$\mathcal{D}$

The first component of computing core chains is building shell–$\mathcal{D}$. To do this, we assume that the density hierarchy has a *maintenance algorithm*. In the case of k-cores, we use [92]. Let the density hierarchy maintenance algorithm be $\mathcal{A}_\Delta$.

The strategy is to begin at time $t = 0$ with a density hierarchy $D_0 = (T_0, M_0)$. However, we store D_0 in a purely functional data structure. Then, we run $\mathcal{A}_\Delta$ to proceed to $t = 1$. This will *modify* D_0 to create $D_1 = (T_1, M_1)$. However, as it is purely functional, the original D_0 will be preserved. D_0 and D_1 become layers in shell–$\mathcal{D}$.

We then need to compute the multiedges for $\mathcal{E}$ in shell–$\mathcal{D}$. It is possible to simply enumerate all vertices and directly compute their location from M_0 and M_1. However, again due to the purely functional data structure, we can record the differences as we build M_1 from M_0. Each difference directly becomes an edge for $\mathcal{E}$.

This process continues for each time point. The full algorithm is given in Algorithm 5.1.

5.4.3 Nuclei Hierarchies for Vertices

Recall that by Definition 5.3, a nucleus is defined as a hypergraph k-core in the (r, s)-hypergraph. This presents a problem when using it to produce hierarchies. We need a

Algorithm 5.2: Computing contracted nuclei

Input: graph $G = (V, E)$, r-cliques R, nucleus density hierarchy $D = (T, M)$
Output: contracted density hierarchy D

```
for v ∈ V do
    X ← ∅
    for r ∈ R(v) do
        X ← X ∪ M(r)
    D ← MergePaths(D, X)                ▷ MergePaths from [92]
return D
```

mapping from every vertex to a node in the hierarchy, so that the vertex movement (Definition 5.7) between parts of the hierarchy can be identified. In the (r, s)-hypergraph, a *vertex* corresponds to an r-clique, which for $r > 1$ is a set of vertices in $\mathcal{G}$.

In order to provide a coherent hierarchy for *vertices*, instead of r-cliques, we *contract and merge* the nodes in the hierarchy tree which have multiple vertices in them. We call the resulting hierarchy a *contracted nuclei* hierarchy. As the size of the tree is very small compared with the size of the graph (for graphs with millions of vertices, trees typically have hundreds or only a few thousand nodes [92]), we compute the contracted nuclei from scratch for each time point. The idea is to simply merge all tree nodes that a vertex is in. The full algorithm is given in Algorithm 5.2.

5.4.4 Computing k-seeded core chains

To compute a k-seeded core chain, we need to follow through layers in $\mathsf{shell}\text{-}\mathcal{D}$ appropriately, ensuring that we always retain the seed set in the current link in the chain. We start with a seed set of vertices, S. We iterate through time linearly. Suppose we are at time t, with the density hierarchy $D_t = (T_t, M_t)$. We then compute the mapping for each of the vertices in S. There are two possible cases: either there is no suitable region, and the chain is finished; or, all of the vertices in the seed set are in some $k' \geq k$ region, and we can continue the chain including the largest such k' region.

The full algorithm is presented in Algorithm 5.3.

Algorithm 5.3: Computing k-seeded core chains

Input: shell$-\mathcal{D}$, k, seed set S, ordered time range T
Output: P, set of sets of dense regions

$P \leftarrow \emptyset$
$C \leftarrow \emptyset$
for $t \in T$ **do**
  $k^* \leftarrow \min_{v \in S} \kappa[k]$
  if $k^* < k$ **then**
    ▷ The region is not dense enough
    $P \leftarrow P \cup C$
    $C \leftarrow \emptyset$
    continue ▷ Stop chain, continue
  $n \leftarrow$ node at level k^* in shell$-\mathcal{D}$ containing any $s \in S$
  if $S \not\subseteq$ vertices(n) **then**
    ▷ Not all seed vertices are in the region
    $P \leftarrow P \cup C$
    $C \leftarrow \emptyset$
    continue ▷ Stop chain, continue
  $C \leftarrow C \cup \{n\}$ ▷ Continue chain with n
$P \leftarrow P \cup C$
return P

Algorithm 5.3 runs in $O(|T|\,\rho S)$. It runs linearly over all time. In each, it first finds the minimum density value κ for each seed set vertex. It then extracts the node at the minimum density for any vertex in the seed set. Next, it needs to check whether all vertices in S are present in the hierarchy. By efficiently implementing inclusion checks with hash tables, it is possible to check this in time ρS, moving up the tree in shell$-\mathcal{D}$ and checking each level in the hierarchy for shell values that contain the element $s \in S$.

5.4.5 Computing k-majority core chains

Next, we describe how to compute k-majority core chains. Instead of starting from a seed set, as in k-seeded core chains, we need to start from a given node and compute the best next node to move to, at the next time point. Starting at the given node, we visit all layer multiedges as we perform a k-limited tree enumeration. The goal is to find the tree node that has the largest number of multiedges that go to either it or any deeper level below it.

Algorithm 5.4: Computing k-majority core chains

Input: shell$-\mathcal{D}$, k, starting node n, ordered time range T
Output: P, set of dense regions

$P \leftarrow \{n\}$
for $t \in T$ **do**
  $C \leftarrow \emptyset$
  $H \leftarrow \mathsf{Heap}()$
  ▷ Starting at n, walk the layer, staying within k
  for *node m at level $\geq k$ connected to n* **do**
    for $(t, t+1, \{m, x\}) \in \mathcal{E}$ **do**
      if $x.k < k$ **then** continue
      x.count $\leftarrow$ number of (m, x) edges
      H.push(x)
  for $x \in H$ **do**
    ▷ Following heap order, largest height first
    $C \leftarrow C \cup \{(x.\text{count}, x.k, x)\}$
    x.parent.count $\leftarrow$ x.parent.count + x.count
    H.push(x.parent) ▷ Heap removes duplicate entries
  if $C = \emptyset$ **then return** P
  $n \leftarrow \arg\max_{x \in C}(x.\text{count}, x.k)$
  $P \leftarrow P \cup \{n\}$
return P

Walking across the layer, we push the number of multiedges into the next layer. Then, we walk backwards down from the next layer, and capture all of the multiedge counts into the nodes at level k.

The full algorithm is presented in Algorithm 5.4.

Algorithm 5.4 runs in $O(|T|\,\rho \log \rho)$, due to the use of the heap. Each tree walk occurs in $O(\rho)$ time and there can be at most ρ elements in the heap. Finally, the algorithm runs for at most T iterations, computing and potentially adding to the result at each time.

5.5 Evaluation

In this section we evaluate the overall core chain approach and explore the quality of results with our two concrete core chains. We consider three use cases: the first is a study of research groups using publication data, which we compute k-seeded core chains on, the second are trust networks of Bitcoin over-the-counter traders, which we compute k-seeded core chains on, and the third is a behavioral study of ants, which we compute k-majority core chains on. We choose k-seeded core chains for the first two as we want to understand the dense regions *the seed set* is in. We choose k-majority core chains for the third, as we want to understand the behavior of a *group* of ants, not any ants in particular.

5.5.1 Identifying Research Groups

We look at the use case of identifying a research group at a university. The objective is to begin with publication data, namely a co-author graph, and identify a research group even as it changes over time. We started with the Microsoft Academic Graph from December 2021 [237]. We extracted papers and authors, and created a clique in the co-author graph for every paper. If a paper had too many authors, we excluded it, as these likely indicate joint work between multiple labs. We explored several thresholds and chose 5 as the threshold, as results did not change much with larger numbers, but the computational complexity increased due to the increase in cliques.

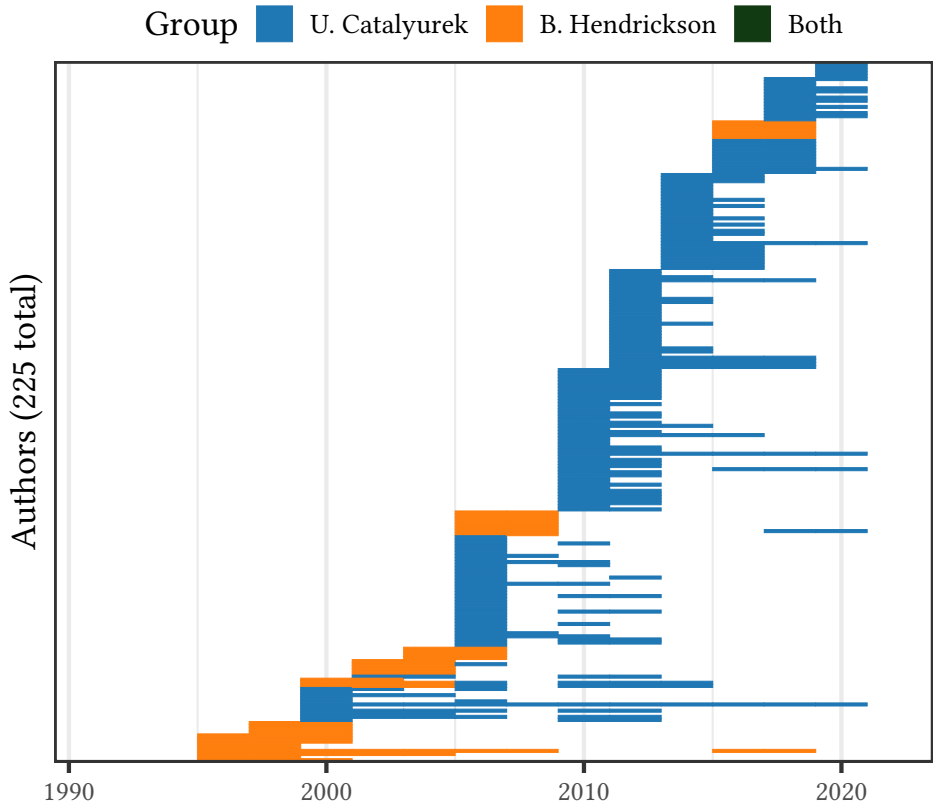


Figure 5.7: Two research groups, in blue and orange, identified through the co-authorship graph using $(3,4)$-nuclei. There are two clearly distinct research groups which evolve over time. No researchers are in both groups concurrently, but several move between groups.

We then computed two k-seeded core chains, one with Ümit V. Çatalyürek as a seed vertex and the other with Bruce Hendrickson. These entries were chosen as they represent distinct, but close, research groups—increasing the difficulty of any approach to distinguish them—and we have ground truth available on lab memberships. We used a two-year age out time for research papers, as two years is a reasonable time for a collaboration resulting in a paper.

In Figure 5.7 we show the $(3,4)$-nucleus k-seeded core chains for both research groups. The groups are distinct, and include membership from one set to another. As an example, Karen Devine is in the same dense region as Bruce Hendrickson, and then between 2004 and 2006 switches into the same dense region as Ümit V. Çatalyürek. Many of the members identified in the research group are Ph.D. students and postdocs. This dense structure is not

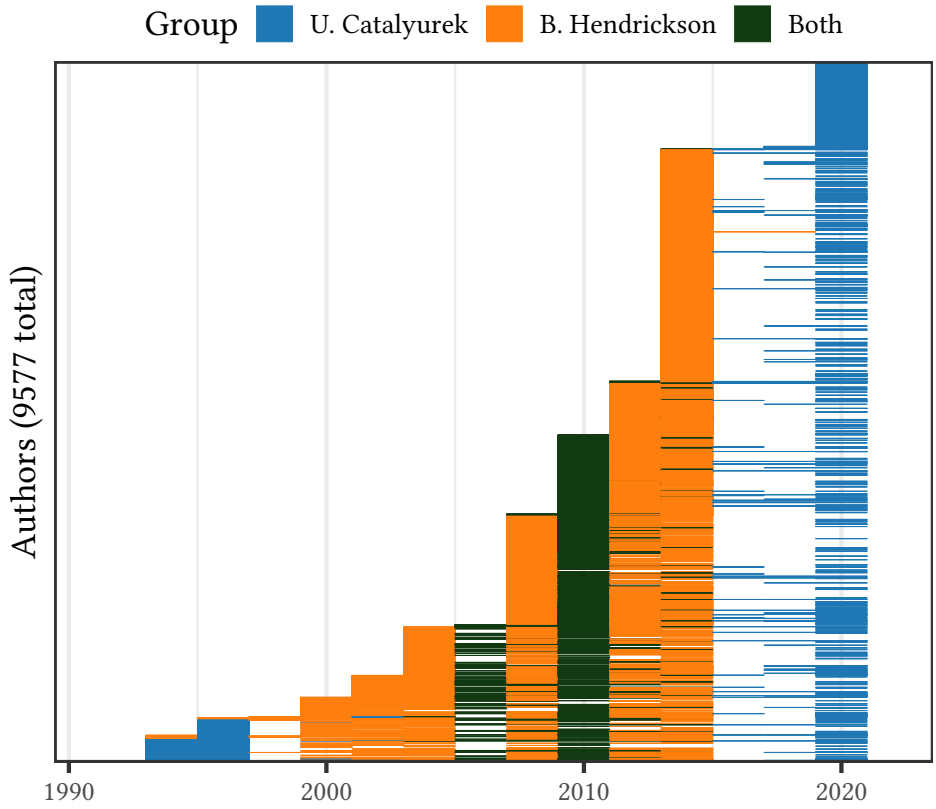


Figure 5.8: Research groups identified through the co-authorship graph using cores ($(1,2)$-nuclei). Compared with Figure 5.7, there is overlap and less coherent group structure, showing the importance of higher order nuclei.

captured simply by co-authors of the seed set, as those contain many more individuals with much looser relationships. As Bruce Hendrickson moved more into upper level management roles after 2010, his research group broadened, and this change to the structure is additionally shown through the core chain.

We next look at lower order nuclei. In Figure 5.8, we show the core structure (that is, the $(1,2)$-nuclei). The structure here is much looser and the number of identified research group members is close to $10,000$. This highlights that higher order nuclei are critical; neither research group had close to this many members. In the middle, around 2010, there are many vertices that are identified as being in both research groups. While there was significant collaboration, the research groups themselves never merged.

In Figure 5.9 we show the same use case but using span-cores [97] instead of core

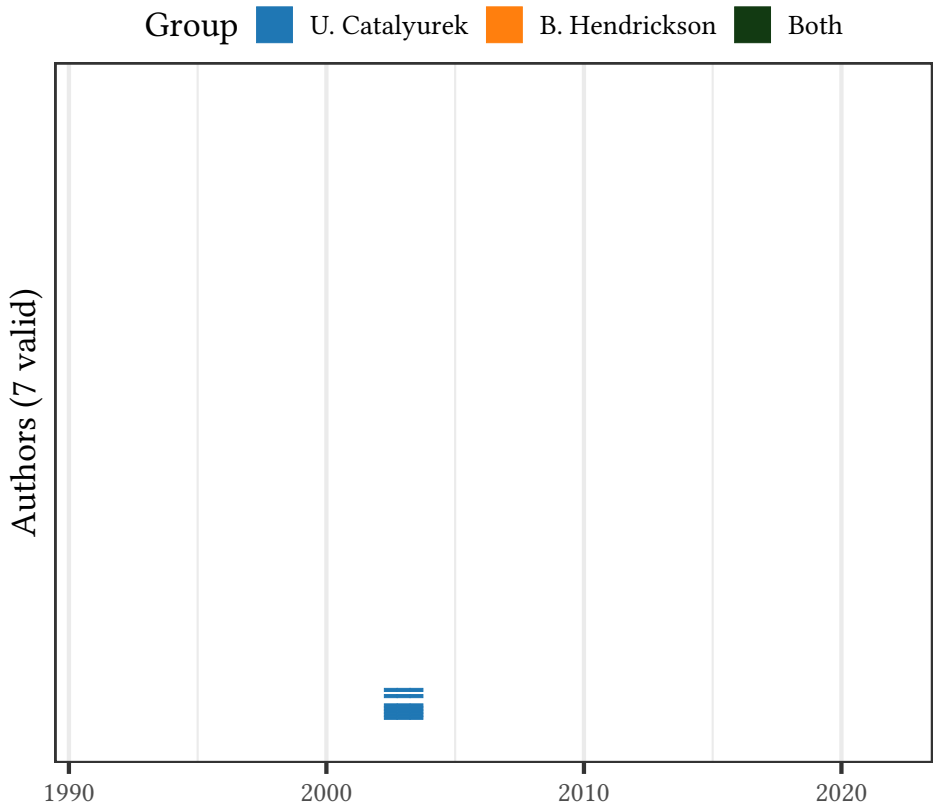


Figure 5.9: Research groups identified with span-cores in the co-authorship graph. Span cores only survive for a very small period of time due to the continual churn in research groups. This similarly holds for other vertex-focused approaches.

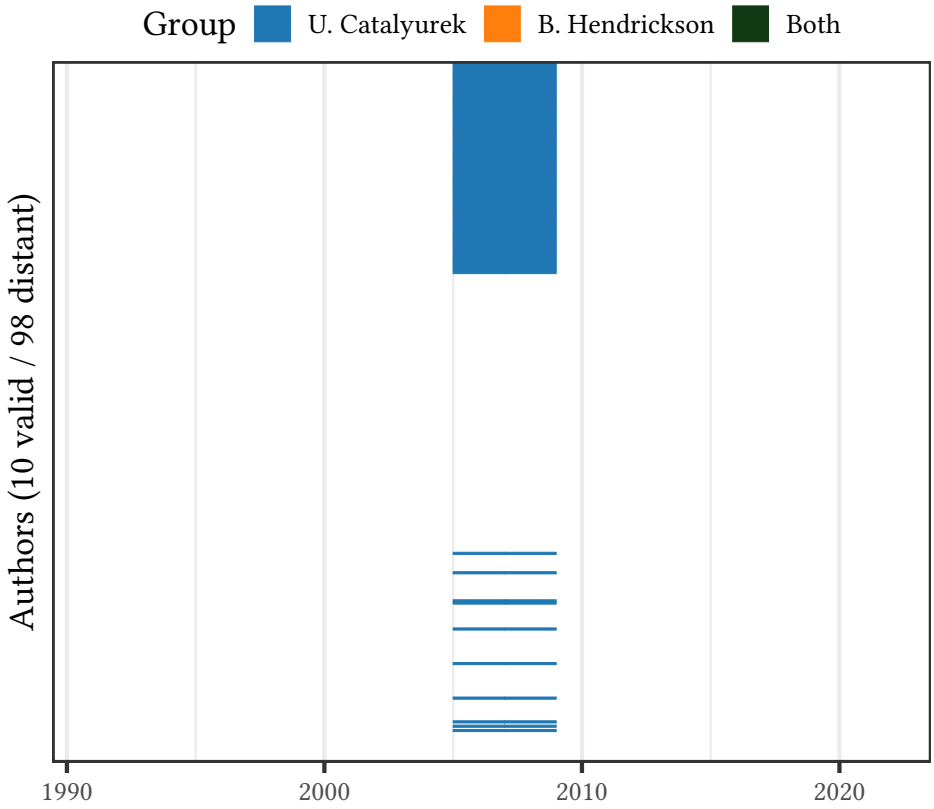


Figure 5.10: Research groups identified with DLCP in the co-authorship graph. There is only one group.

chains. Unfortunately, as a vertex-focused approach, there are no dense regions that last very long. Research groups are too volatile. As such, close to one time point, even at different age out periods, is returned. Vertex-focused approaches are not able to identify such continuously changing research groups.

In Figure 5.10 we show the results with diversified lasting cohesive subgraphs [167], denoted DLCP. After performing a parameter sweep for k, r, and σ, this was the best available result, with $k = 4$, $r = 2$, and $\sigma = 5$. Here, DLCP again has a *vertex-focused* result—there is no temporal change in the found subgraph. There are 10 authors that were identified by k-majority core chain as well during the time, and those appear to be in the correct research group. However, there are 98 authors that are included in the dense, cohesive region that would not be considered part of the research group: for example R. Shuttleworth, who has no publications with the group or its close collaborators.

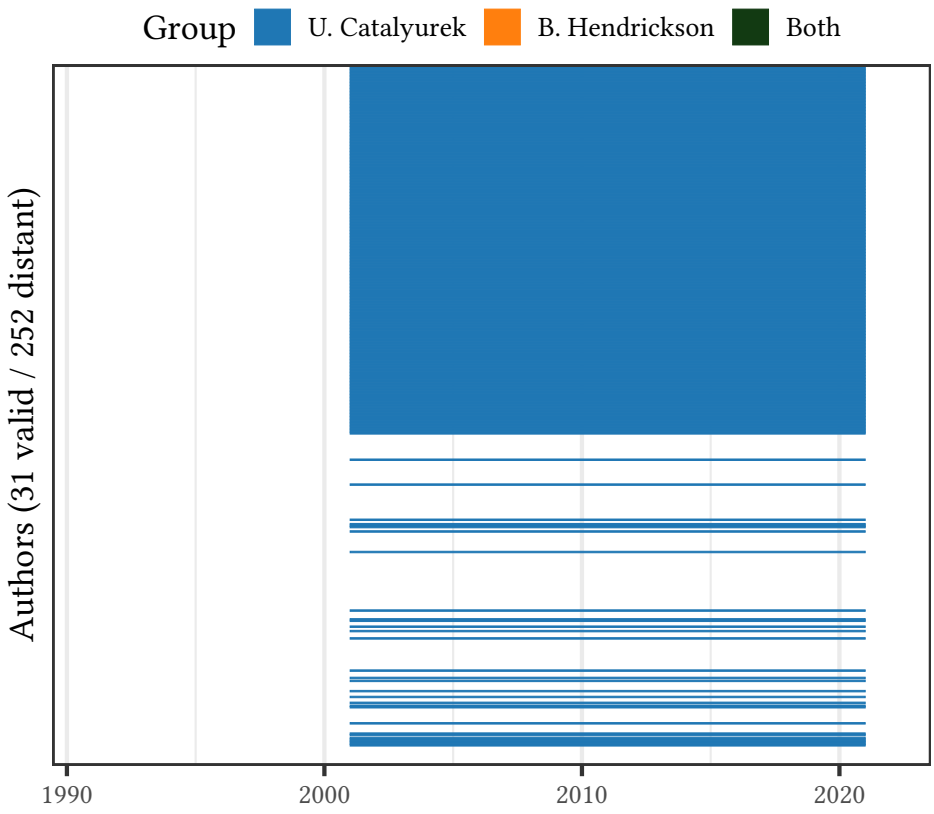


Figure 5.11: Research group identified with MBC in the co-authorship graph. The group contains a large number of authors that are not in the same research group.

In Figure 5.11 we show the results of computing maximal bursting cores (MBC) [207]. We again performed a parameter sweep, and show the best result here. Unfortunately, the bursting core period covers almost all of the range. There are 252 authors there were identified as being in the core that are quite distant from the target research group, including members such as O. Ohia, publishing in specific physics fields. Importantly, this approach similarly cannot capture temporal regions that are changing over time, as the focus is on static vertex sets.

Of the baseline algorithms, span-cores produce the best results. As such, in the following we focus on comparing against span-cores.

5.5.2 Bitcoin Trust Network

We next look at a trust network of over-the-counter (OTC) Bitcoin users [150, 149]. Here, data was collected from the #bitcoin-otc web of trust network. We call fraudulent users bad and legitimate users good. Each vertex in the graph is a user. Edges represent a *vote* of a user's understanding of whether another user is bad or good. Specifically, each edge is temporal, *weighted*, and *directed*. The source represents the user making the trust evaluation, the destination represents the user being evaluated, and the weight (from -10 to 10) reflects the trust evaluation itself. A negative value means the source recommends not trusting the destination and a positive value means the opposite, and the farther the evaluation from 0 the higher the source's confidence in the rating. Note that as this is an anonymous system it is vulnerable to Sybil attacks [63] and so it is not possible to simply sum up the negative or positive values: any number of users can easily be created and these users can vote, so a very large positive value is not necessarily meaningful.

Kumar et al. [150, 149] provide algorithms and approaches that identify the bad users with high accuracy and recall. We are not developing an approach to classify bad users; instead, we want to find *cohesive groups* that are good or bad. A cohesive bad group, for example, may represent a single entity that is creating bad OTC users that all support each

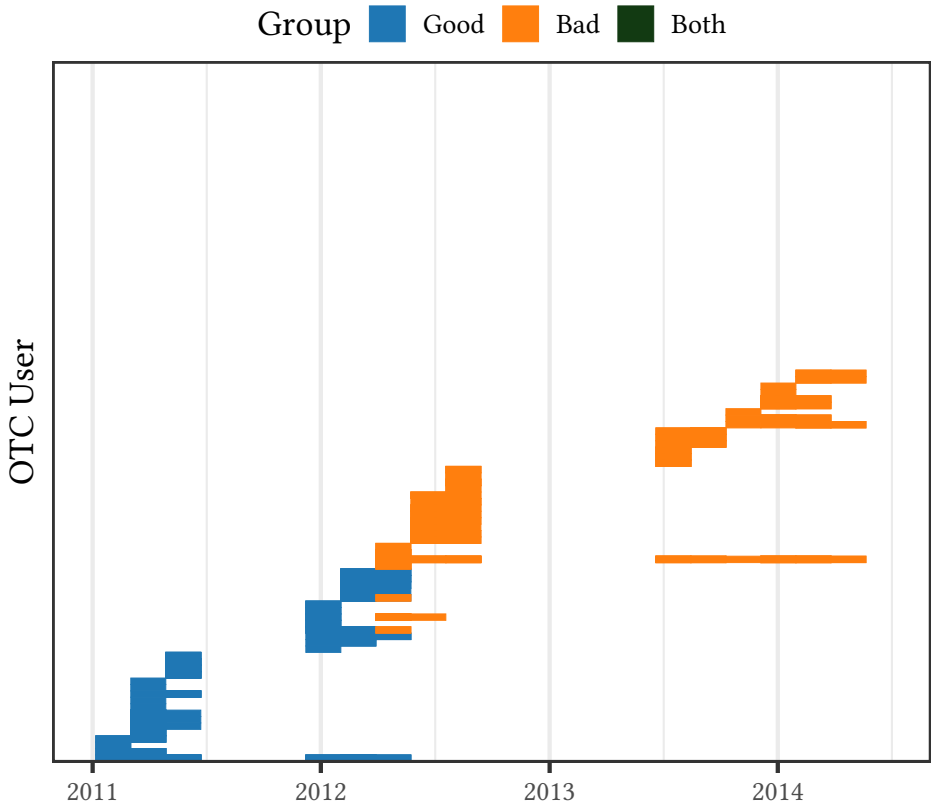


Figure 5.12: The results of running $(3,4)$-nuclei k-seeded core chains for a good user (user 1) and a bad user (user 905). There are two distinct dense regions. User 905 is active before 2012, however they are not part of any dense region. It is possible they attempt to correct their rating with fraudulent bots after 2012.

other. We assume that we start with a seed user, who is known to be either good or bad. In our experiments, we use the ground truth computed from [149] to choose a seed good and bad node. To exercise our approach, we choose arbitrarily chose the first two users that we found that were (1) active in overlapping time ranges and (2) had ground truth. Our seed vertices are the user 1 (good in the ground truth, also the site creator) and user 905 (bad in the ground truth, and potentially built up for a large scam over time).

As our approach only operates on simple graphs, those that are undirected and without weights, we only keep positive weighted edges and symmetrize the graph. We choose an age out period of two months.

The results of running $(3,4)$-nuclei based k-seeded core chains with the two different seed sets are shown in Figure 5.12. There are 30 identified good users and 34 bad users.

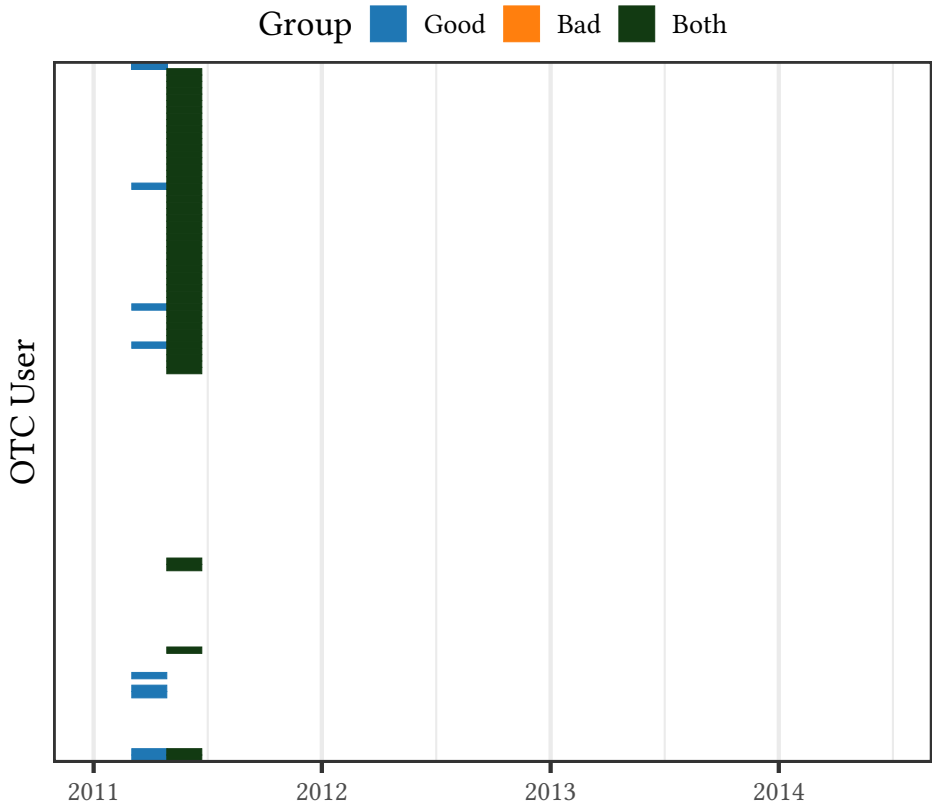


Figure 5.13: The densest span-cores returned with user 1 (good) and user 905 (bad). The dense span-core with user 905 also had user 1, and so is marked as in both group.

Evaluating against the ground truth from [149], there is one false positive good user and three false positive bad users. We have 12 true positive good users and 3 true positive bad users. This results in a precision of 0.92 for good and 0.5 for bad. Note that there are only 316 ground truth entries out of 5881. There are two clearly distinct core chains, and both the good and bad core chain changes over time as users become inactive. The false positives for bad users occur early on, potentially even when user 905 was acting good.

In Figure 5.13, we show the results of the same experiment running span-cores. We extracted the densest regions with both the seed 1 and the seed 905. As expected, the longest densest region was only one time point in both cases, as dense regions undergo continual internal change. For all tested seed users, the results are similar: there is too much churn to find meaningful temporal regions with vertex-focused approaches.

5.5.3 Tracking Ant Behavior

In [190, 212] ants are placed in a controlled environment and given unique tags. An infrared camera is used to capture the position and movement of all of the ants. If two ants move close together, are facing each other (within a certain angle range), and then move apart, the behavior is categorized as an interaction. In [190, 212], the ants were classified both spatially and through such interactions, and the research teams uncovered that ants both have defined roles and change those roles over their lifespans. While queen ants can live for decades, other ants tend to have short life spans—measured in days or weeks [190].

A *nurse* is an ant that stays close to the queen and assists with new ants and management of the queen. A *forager* is an ant that leaves the nest, explores, and attempts to bring food back into the nest. In [190, 212] these roles were identified as the two main ant roles. The researchers used a mixture of spatial tracking and interaction tracking to identify the two roles. The data in [190] is publicly available at a granularity of one day, however the ground truth manual analysis was only performed on ten day intervals.

In Figure 5.14 we show the results of finding a k-majority core chain on the interaction data of an ant colony over time. The experiment was run for 41 days. In orange we show when an ant is part of the k-majority core chain built from $(2,4)$-nuclei and starting with nurse ants, and in blue we show ants classified as nurses by [190]. We chose an age out period of 1 day. Green shows ants that were classified by both as nurses. While k-majority core chain has daily resolution given the data, the baseline ground truth study was broken into 10 day chunks to try to stabilize the community detection and to ease manual analysis. In the original work, an *interaction per day threshold* was used to filter the graph and only include edges with at least some threshold number of interactions per day. We perform the same preprocessing and use a threshold of 12. To compare results, we also smooth the results to 10 day chunks by looking for the inclusion of an ant in the k-majority core chain within the 10 days to mark the ant. There are 59 overlapping blocks, 362 blocks that are both empty, 16 in [190] but not in the k-majority core chain, and 15 in the k-majority core

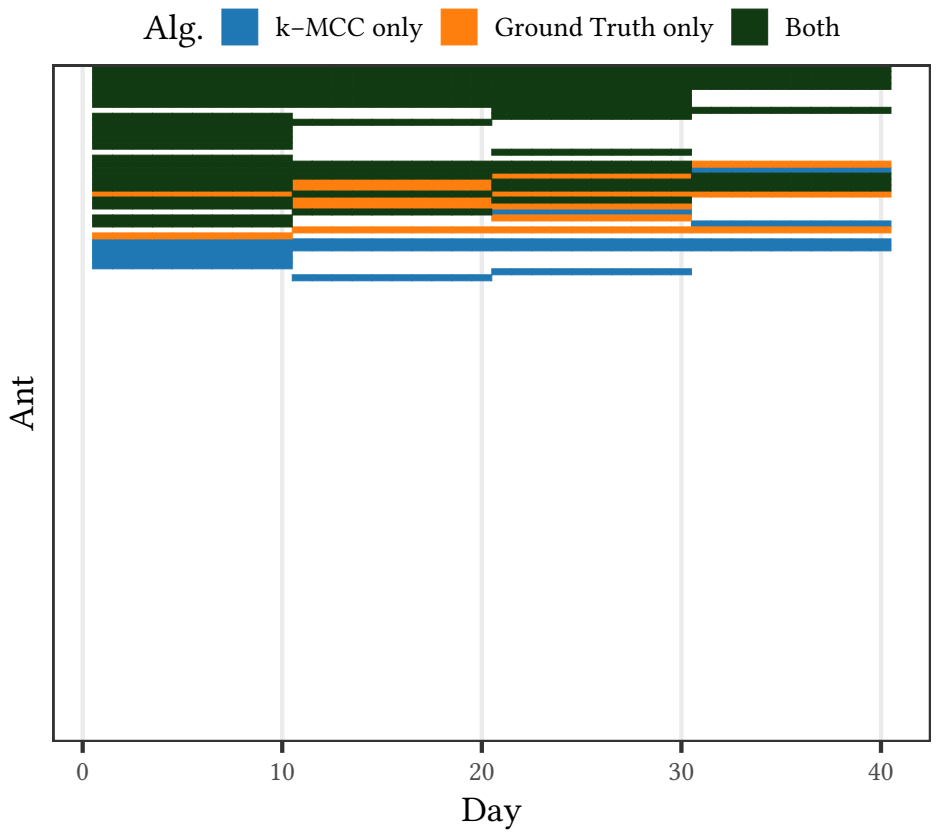


Figure 5.14: The performance of k-majority core chain for finding nurse ants, starting with the $(2, 4)$-nuclei that contains the largest number of nurses initially, compared against the manual temporal construction of ant behavior by [190], with jointly labeled ants in green. The k-majority core chain follows the temporal behavior for many of the ants. White blocks indicate ants that are not categorized as nurses by both algorithms. Ideal case would be entirely green and white.

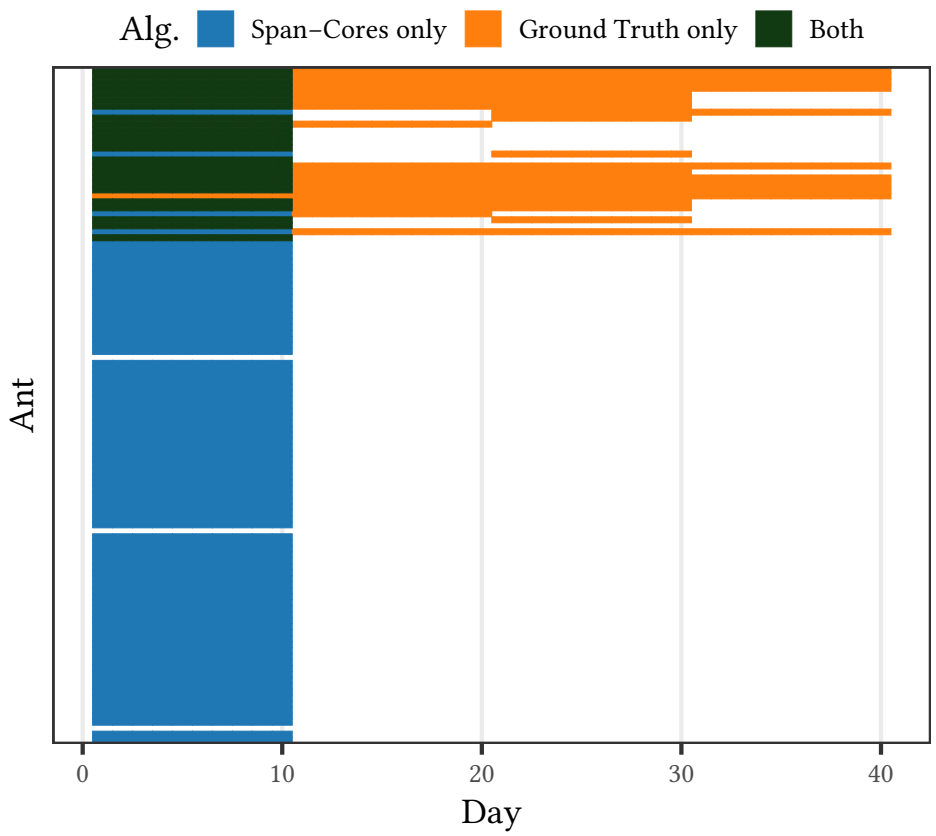


Figure 5.15: The same experiment as Figure 5.14, but showing span-cores instead. There are no span-cores with nurses that survive for beyond one temporal block. Ideal case would be entirely green and white.

chain and not in [190].

In Figure 5.15 we show the results of computing maximal span-cores [97]. We identify this problem as stemming from the inherently temporal nature of the dense regions, resulting in vertices coming and leaving dense regions. There are 24 overlapping blocks, 293 blocks that are both empty, 85 in [190] but not in the span-core, and 50 in the span-core but not in [190]. In particular, the span-core exists for only one temporal period, making it ineffective as a temporal dense region. Similar results occur for [207]. In [167], only one bursting diversified r-core was found, from time 1 to 2, with all ants included. Unfortunately, all available vertex-focused approaches are not able to find any temporal dense regions.

We chose $(2, 4)$-nuclei as it robustly provided an interesting hierarchy. In many cases,

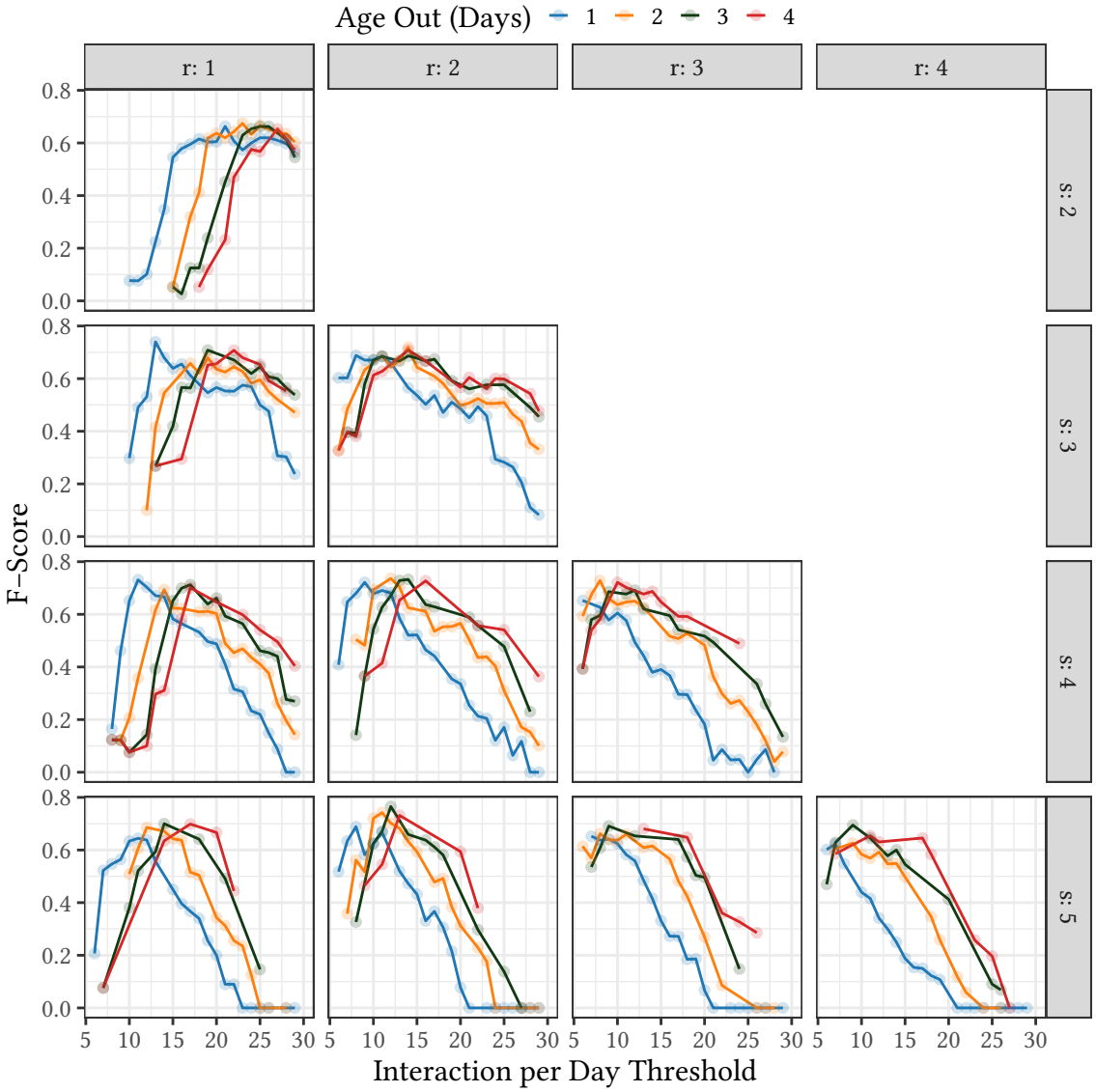


Figure 5.16: The F_1-score for separating nurse ants from other ants with different nuclei, interaction per day thresholds, and age out times for interactions. The maximum F_1-score is with $(2, 5)$-nuclei, a 12 interaction threshold, and an age out of 3 days. Span-cores achieve an F_1-score of 0.26.

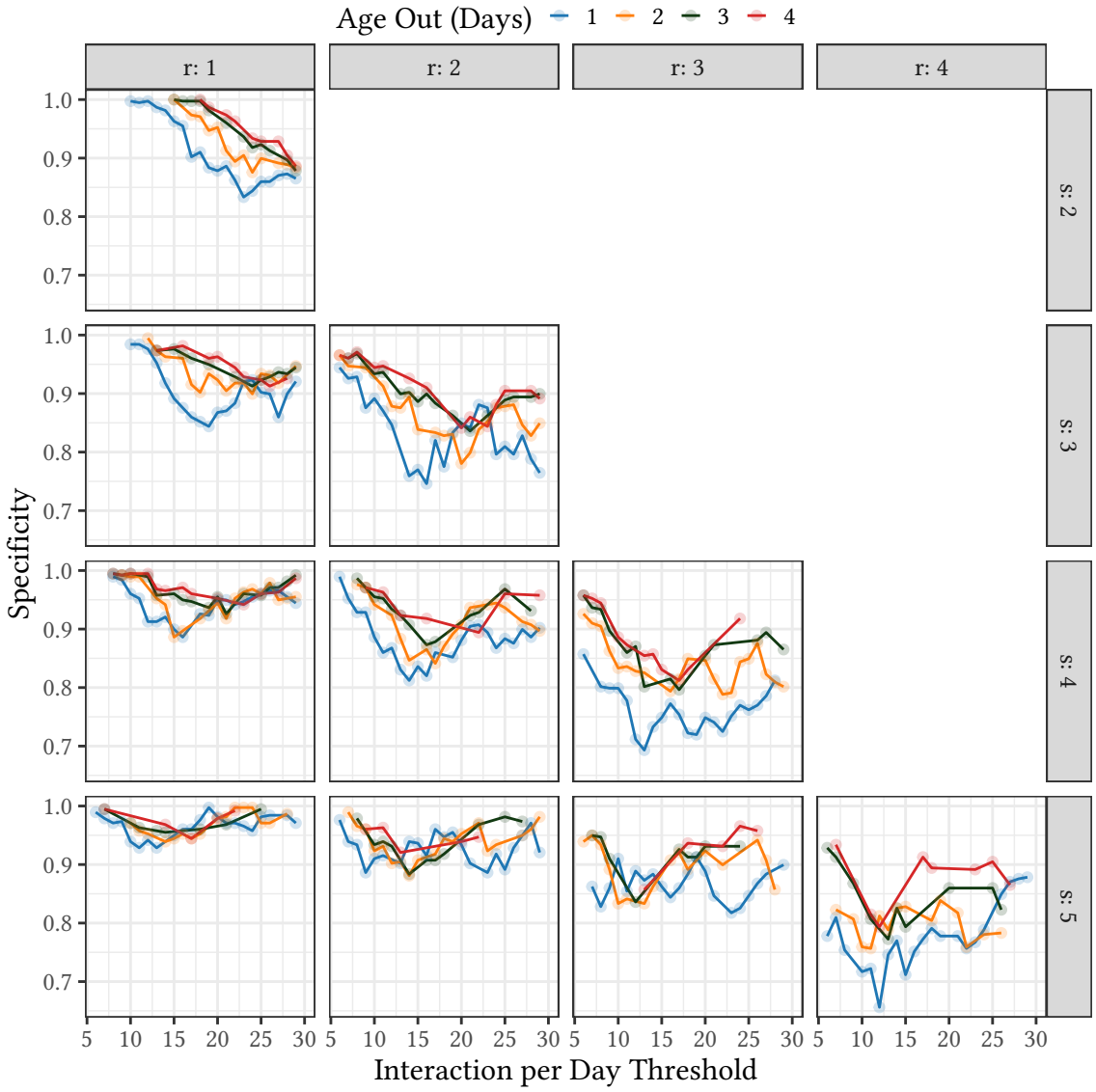


Figure 5.17: The specificity for separating nurse ants from other ants. This is important, as we want to avoid labeling everything as being in the dense region. Span-cores achieve a specificity of 0.76.

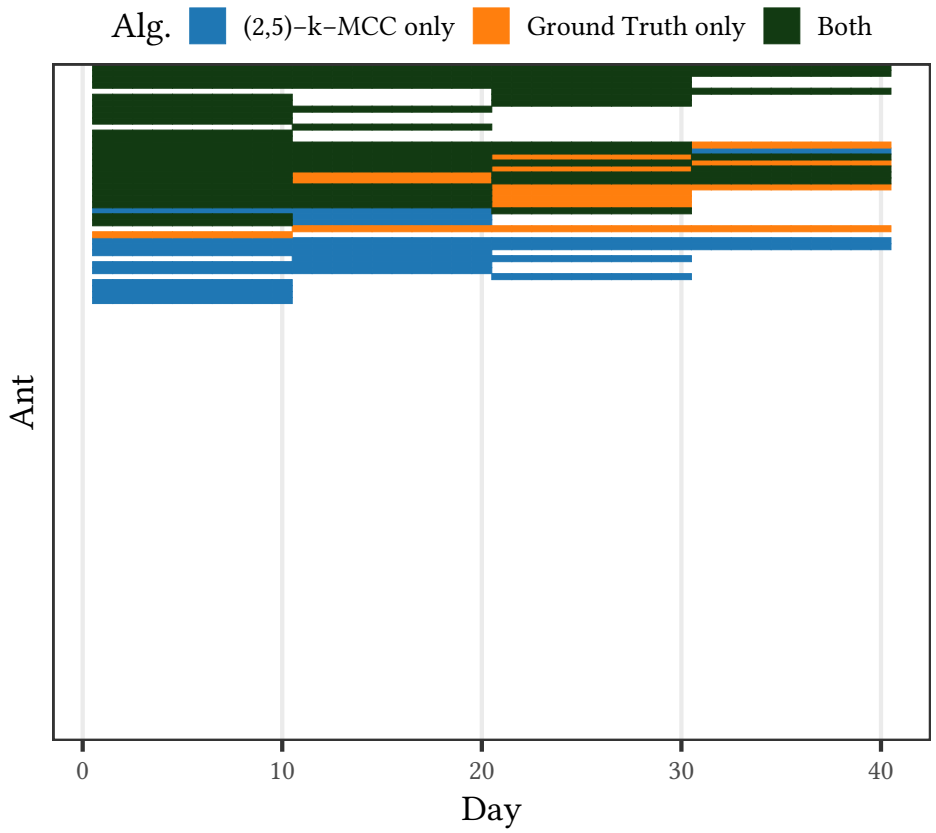


Figure 5.18: The performance of k-majority core chain for finding nurse ants, starting with the $(2, 5)$-nuclei, which has the highest F_1-scores and higher specificity than $(2, 4)$-nuclei based core chains.

$(1, s)$- and $(2, 3)$-nuclei provide a single dense region, which is not able to tease out different characteristics, such as groups of ants. We show the F_1-score of different nuclei, age out periods, and interaction thresholds per day in Figure 5.16. The highest F_1-score is 0.76, with $(2, 5)$-nuclei, an 12 interaction per day threshold, and an age out period of 3 days. In Figure 5.14, the $(2, 4)$-nuclei we show with a 12 interaction threshold and an age out period of 1 day has an F_1-score of 0.68. From the results the best scores tend to be in the threshold range from 10 to 15, with $r = 2$. With a longer age-out period, in many cases a higher threshold is required for the same F_1-score. Additionally, as r and s increase, the threshold tends to need to be lower, meaning more data is required.

From the results, having $r = 1$ is less useful for this dataset, and at $r = 4$ the quality appears to begin to drop off. In Figure 5.17 we show the specificity, which focuses on the

true negatives. We do not want to classify everything as being in the dense region, as that removes the purpose of finding a dense region. From these results, it becomes clear that the $(2,5)$-nuclei perform well, while retaining high F_1-scores. Span-cores achieve a specificity of 0.76, which again is below almost all nuclei core chain variants. In Figure 5.18 we plot the $(2,5)$-nuclei k-majority core chain along with the provided ground truth. Similar to with $(2,4)$-nuclei, many of the temporal dynamics of individual ants are captured. Even more so than with $(2,4)$-nuclei there is evidence of a main result of [190], showing that ants start as nurses and then move away as time goes on.

Exploring a systematic method to select the appropriate nuclei for a core chain is an important avenue for future exploration. An effective heuristic could potentially incorporate preprocessing to choose suitable thresholds and nuclei together. Additionally, developing approximations to address even higher order nuclei would likely improve the results further.

5.6 Summary

We address the increasingly important problem of identifying dense regions in temporal graphs, those with historical information. We identify that vertex-focused definitions are not appropriate when the temporal, dense regions themselves are changing. We present a new temporal dense region, called core chains. Core chains are defined as paths between nodes in the dense hierarchy at different times in the graph. We propose two concrete core chains, a k-seeded core chain which ensures that seed vertices are always in the chain, and k-majority core chains, which follow the majority of vertex movements between points in the dense hierarchies. We implement core chains using nuclei, a generalization of graph cores that include richer and more stable hierarchies. Together, we show that these two core chains are able to find useful and important regions through three use cases: co-author graphs that can identify research groups, trust networks that can find cohesive entities of good and bad users, and ant interaction graphs that can uncover behavior of ants over time. We achieve F_1-score improvements of up to 0.5 higher than prior state-of-the-art results.

CHAPTER 6

LOADING AND SAVING MASSIVE GRAPHS

There is significant work developing highly optimized sparse linear algebra and graph systems, including competitions for graph kernels [219] and sparse machine learning [139], programming models and corresponding high-performance libraries [234, 23, 199, 182], and full graph databases [27]. These all focus on providing high-performance kernel runtimes for a variety of datasets. In this chapter, we are not providing yet another computational graph library or a new graph programming model. Instead, we address an important—yet largely overlooked—aspect of *using* and *developing* sparse graph and matrix systems: the input and output (I/O). Our focus is on shared-memory multi-core servers. We show that graph I/O is frequently the single slowest factor in the end-to-end performance of otherwise fast computations. On billion-scale graphs, reading a graph easily takes over 2000× longer than running a computational kernel.

In many cases implementations have stuck with sequential I/O, presumably under a longstanding belief that parallel I/O is not achievable without a dedicated parallel I/O system. In the literature, graph and matrix I/O times are rarely reported and, hence, not highly optimized. While it is the case that SATA serializes disk access [246], implementing only sequential I/O misses three major and common opportunities for parallelism. First, a hardware Redundant Array of Inexpensive Disks (RAID) controller can read from multiple drives over separate SATA connections in parallel [203]. Second, Non-Volatile Memory (NVM) is now widely deployed [133] and provides parallelism through both SSDs and the NVM express (NVMe) interface to motherboards [246]. Last, file systems themselves include tuned and effective caches, supporting parallel reads and writes [181]. In these cases, reading and writing in parallel can provide a significant end-to-end improvement for applications.

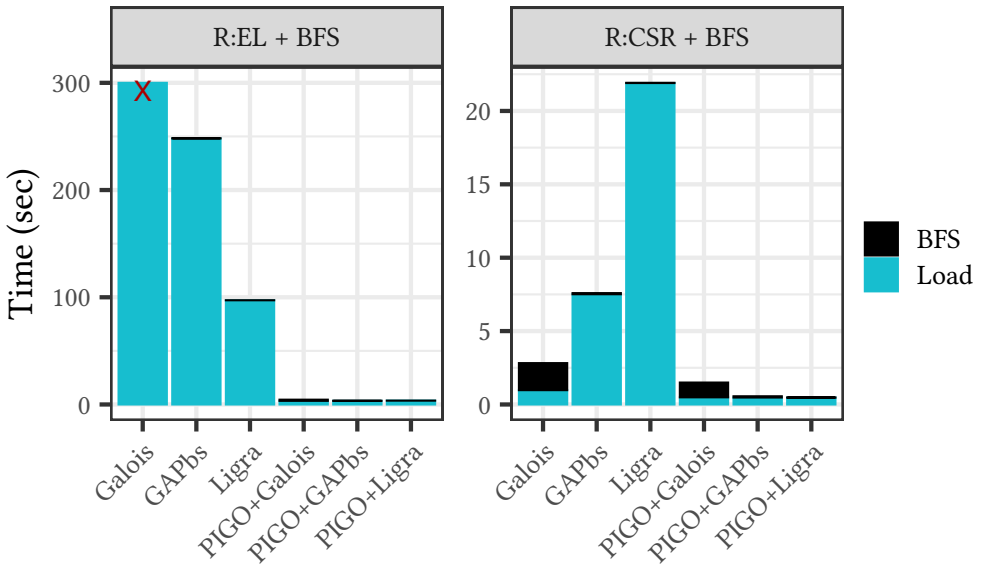


Figure 6.1: Edge list (R:EL) and binary (R:CSR) read times and BFS runtimes for com-Friendster, a social network graph with 65 M vertices and 3.6 B edges. Note the differing scales between R:EL and R:CSR. Using PIGO significantly improves end-to-end runtimes for all systems, bringing them much closer to kernel runtimes. Galois did not finish R:EL within several thousand seconds.

Our goal is to *remove the burden of building efficient I/O*. We target two types of users: the researcher, who is developing a single graph kernel and does not care about the production readiness of the code; and the developer, who is building a graph library for production level end-user use.

Importantly we need to be *useful* for researchers who want to test out a kernel idea quickly without buying into a large, complex graph system, complete with a steep learning curve and powerful, yet complex programming models. We do not want researchers to have to stick with a particular graph format, spend energy converting between them, or designing ad-hoc binary formats with potentially subtle errors. At the same time, we seek to provide *best-in-class performance*.

To achieve our goal, we provide PIGO[1], a small, C++11 header-only library. PIGO takes in a filename and returns the graph loaded into memory. Compared against optimized and parallel graph loading in state-of-the-art libraries or simple, ad-hoc loading, we show

[1] Available at https://github.com/GT-TDAlab/PIGO.

that using PIGO can quickly increase both productivity and end-to-end performance. In Figure 6.1, we show three leading graph libraries [234, 23, 199] running breadth-first search (BFS) on a large graph. When running without PIGO, it takes over one hundred seconds to load from ASCII edge list file and 0.01 seconds to run. With PIGO, the loading time is reduced to 3.5 seconds (from an edge list) or 0.5 seconds (from a binary compressed sparse row).

PIGO enables dynamic graph algorithms to quickly load edge list streams and graph checkpoints, ultimately reducing the query staleness as end-to-end runtimes increase.

6.1 PIGO I/O Library

6.1.1 Requirements

PIGO is built to satisfy the following requirements.

Requirement 6.1. *Fast enough to make effective use of modern hardware and operating system's parallel I/O performance.*

Requirement 6.2. *Support for common front-end and back-end graph formats, removing the need for slow and ad-hoc preprocessing, including weights, symmetry, and directedness.*

Requirement 6.3. *Easy and useful integration with both graph analysis systems and one-off graph and matrix programs.*

We address these requirements through our design. First, we read in parallel and we decode bytes directly into integers, floats, comments, or spaces. Our library was developed with performance as a goal and written carefully as such. For example, we limit unnecessary runtime code through template parameters which importantly avoids branch mispredictions.

As a library focused solely on I/O, we are able to address Requirement 6.2 by ensuring we can quickly add new formats. Front-end formats are graphs that are stored on disk,

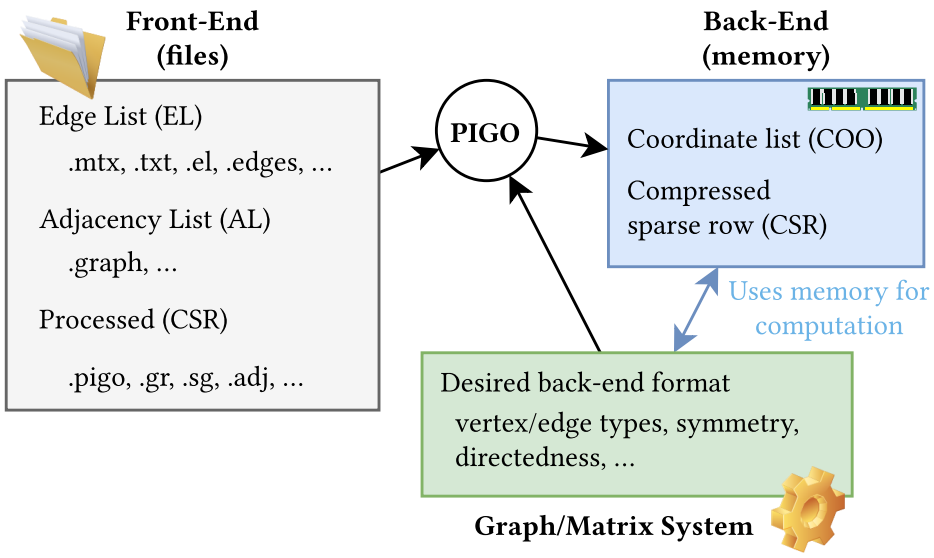


Figure 6.2: As an I/O library, PIGO takes a desired back-end configuration from the computation system and transforms the front-end appropriately.

downloaded, and used in graph pipelines. The two main formats are *edge lists* (`EL`) and *adjacency lists* (`AL`), both of which are ASCII encoded. On the back-end, we currently support two structures. The first is a *coordinate list* (COO), which stores non-zeros with their explicit row and column coordinates, and the second a *compressed sparse row* (CSR), which stores non-zero elements in a single contiguous block of memory and a separate offsets block that stores the beginning of rows. At the moment we support several basic preprocessing steps such as removing self loops and symmetrizing the graph. Current and future work includes more robust preprocessing and support for additional formats. Figure 6.2 shows PIGO's high-level design meeting this requirement.

Finally, we build PIGO to be a C++11 header-only library, allowing it to be easily integrated into projects and systems. The programming interface is designed to be simple to learn and quick to use, as shown in Section 6.1.3. For further usability, we have an experimental port which wraps PIGO into a shared object, making it potentially available to applications in C, Rust, Python, and more.

6.1.2 Overview

PIGO reads *in parallel* from a variety of ASCII files along with custom binary files. Reading binary files is straightforward: the size information for data can be quickly encoded and read from the file header, and the transfer itself consists of parallel and independent `memcpy` calls. ASCII files, however, are the standard file format for both large and small sparse matrices [143] and graphs [214].

PIGO handles ASCII files with two passes. First, PIGO loads the input file into memory via `mmap`[2]. In the first pass, the structure is read out in parallel. That is, the number of spaces, newlines, and integers are counted—while ignoring comment lines and end-of-line comments. After this, memory is allocated and a prefix sum is performed so each thread knows its position in the back-end memory to write to. The second parallel pass then iterates over the file again, parses the integers and copies out the data.

6.1.3 Application Programming Interface

PIGO is used by declaring a *back-end* format and providing a filename as input. PIGO then loads the file in parallel and converts it appropriately into the requested back-end structure. Parameters, such as the data types to use inside the matrix and preprocessing flags, such as whether to symmetrize the matrix, are given as template parameters.

In Figure 6.3 we show the main concept of our API. Our complete API is documented in our repository.

`Label` contains the type of the row or column (or vertex) labels. In many cases, it can be a 32-bit unsigned integer. The `Ordinal` type contains the type that will count (hence ordinal). If the number of edges is large, this may need to be 64-bit. The `Storage` types indicate how PIGO should allocate the memory. PIGO supports raw pointers (`T*`), vectors (`std::vector<T>`), and shared pointers (`std::shared_ptr<T>`). Finally,

[2] `mmap` is a POSIX-compliant call to place the file contents into memory, making the whole file available as a `char*`.

```
COO<Label, Ordinal, LabelStorage, Flags> COO { filename };
void COO.save(filename);
LabelStorage COO.x();  // Get row labels
LabelStorage COO.y();  // Get col labels

CSR<Label, Ordinal, LabelStorage, OrdinalStorage, Flags> CSR {
    filename };
void CSR.save(filename);
LabelStorage CSR.endpoints();
OrdinalStorage CSR.offsets();

Graph { filename } : CSR;
EdgeIt Graph.neighbors(Label v);

Matrix { filename } : CSR;
RowIt Matrix.row(Label r);
```

Figure 6.3: The high-level API for PIGO.

```
#include "pigo.hpp"
#include <iostream>
int main(int argc, char** argv) {
    pigo::Graph g { argv[1] };
    for (auto n : g.neighbors(123))
        std::cout << n << std::endl;
    return 0;
}
```

Figure 6.4: An example program using PIGO with default template values.

the `Flags` are used to indicate various preprocessing steps, for example to symmetrize the file or to remove self loops.

6.1.4 Example Programs

Here we show two examples. The first is a simple standalone program that a researcher might write to develop a new graph kernel. To install PIGO, only a single `pigo.hpp` file is needed. The complete example can be seen in Figure 6.4.

For the next example, we show how to extend Ligra [234] to PIGO+Ligra, allowing it to take advantage of significantly improved *binary and ASCII* loading. The function

```
template <class vertex>
graph<vertex> readGraphFromFile(char* fname, bool, bool) {
    pigo::Graph<uintE, uintT,
            uintE*, uintT*> g {fname};
    long n = g.n();
    long m = g.m();
    uintT* offsets = g.offsets();
    uintE* edges = g.endpoints();
    // Continue with remaining Ligra code
    ...
}
```

Figure 6.5: A replacement `readGraphFromFile` function for Ligra. This will cause Ligra to read with PIGO, resulting in PIGO+Ligra.

`readGraphFromFile` is replaced with the code in Figure 6.5. As PIGO takes care of reading the file, and can handle preprocessing steps, all that needs to occur beyond the PIGO call is building Ligra's vertex objects.

6.1.5 Algorithm Details

There are two main problems reading ASCII files. The first is if we evenly partition the data into chunks, the partitions may not line up on clean integer boundaries. The second is that the destination to write to in memory is not known apriori. We solve the first problem by adjusting the start and end boundaries locally for each thread. Concretely, each thread finds either the next newline or the next integral character and sets that as the thread data boundary. For P threads, this can result in up to P additional reads of bytes, however for any reasonable file the number of bytes overlapping between segments is a small constant. Solving the next problem is done via the two passes described in Section 6.1.2. Note that these passes are done in parallel. We parallelize with OpenMP.

To make this more concrete, we present the core idea for reading `AL` in Algorithm 6.1. A visual example is shown in Figure 6.6. `EL` reading is similar but more simple, as N only needs to contain newlines. Binary reading and writing simply reads or writes in parallel at byte boundaries.

Algorithm 6.1: The core idea of the `AL` reading.

Input: memory mapped file F

$N[1, \ldots, \text{num threads}] \leftarrow \langle 0, 0 \rangle$
for *chunk* $c \in F$ **do in parallel**
  $t \leftarrow$ thread ID
  $N[t] \leftarrow \langle$ integer count , newline count $\rangle$
allocate offsets, endpoints
prefixSum(N)
for *chunk* $c \in F$ **do in parallel**
  $\langle e, v \rangle \leftarrow N[t]$
  foreach *integer in* c **do**
    endpoints$[e] \leftarrow$ data
    if *passed newline* **then**
      offsets$[v] \leftarrow e$
      $v \leftarrow v + 1$
    $e \leftarrow e + 1$

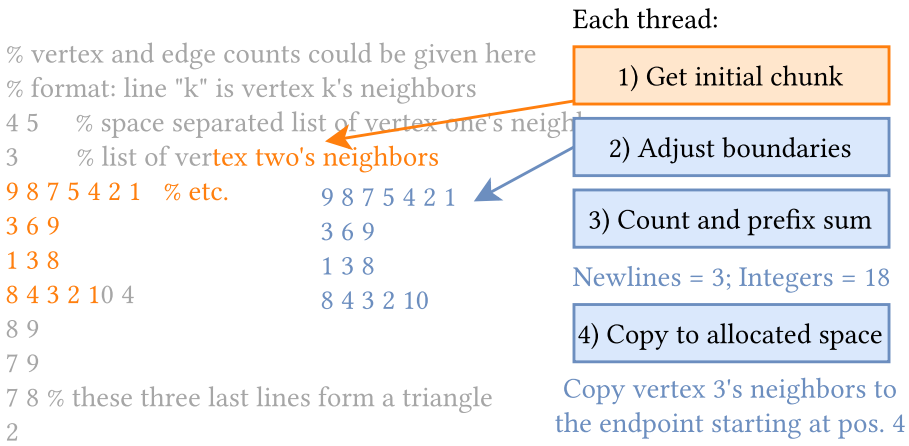


Figure 6.6: An example reading an `AL` in PIGO.

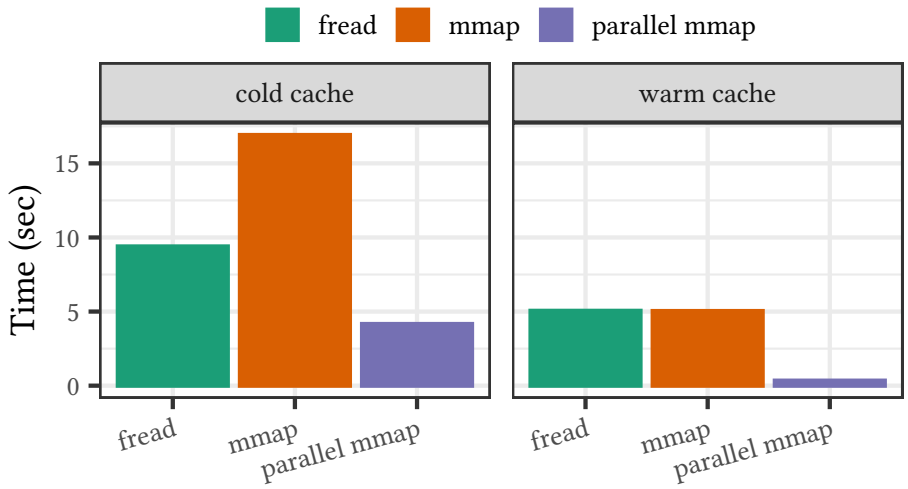


Figure 6.7: A microbenchmark showing the worst-cast (cold cache) read times of 10GB of random data with a parallel `mmap` and sequential strategies.

6.2 Experiments and Results

In this section we present experiments and results. We want to demonstrate that PIGO can read ASCII and binary files significantly faster than known graph systems, providing significant end-to-end improvements.

As exemplar graph systems, we use Ligra [234], GAPbs [23], and Galois [199]. We compiled with GCC 9.1.0 and ran on a machine with 1TB of RAM, an Intel DC P3700 2TB NVMe SSD connected with PCIe 3.0, and two 18-core Intel Xeon E5-2695 v4 CPUs at 2.10 GHz. We ran Ubuntu 16.04 with ext4. Our datasets are from the Network Repository [214] and SuiteSparse [143].

In Figure 6.7 we show the potential parallel improvement for reading a large binary file starting from an empty, or *cold cache*. A warm cache is common when running multiple experiments or using a freshly downloaded file. With NVMe, even a cold cache has parallel potential. When the cache is cold, reading in parallel is around twice as fast as reading sequentially. However, with warm caches, the memory of the system becomes a factor. Here, reading in parallel is 15 × faster and gets close to the system memory bandwidth.

We stress there is a significant room for I/O improvement beyond simply reading bytes

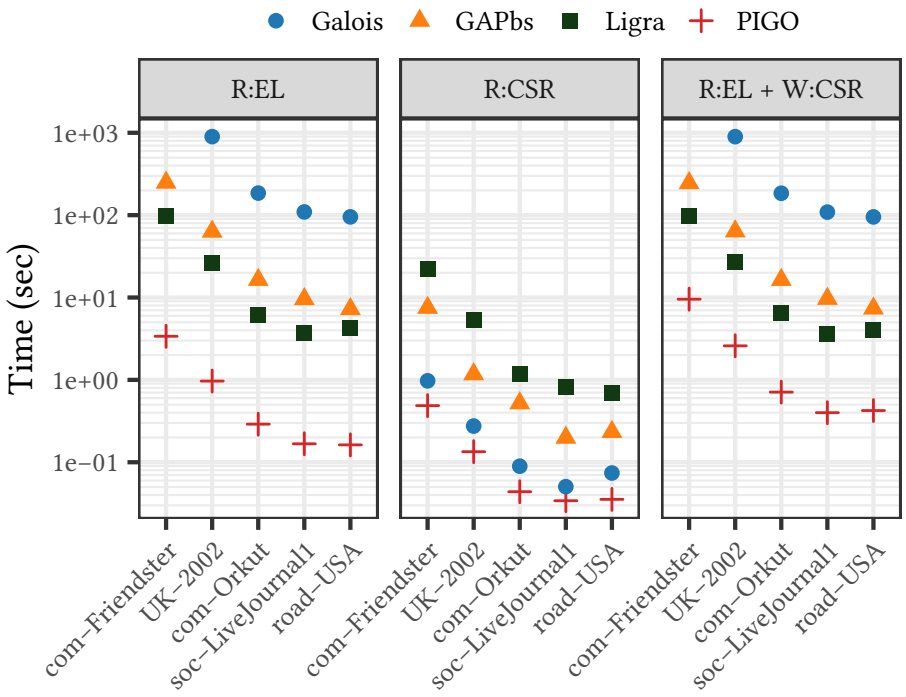


Figure 6.8: I/O times for tested graphs with GAPbs's I/O, Ligra's I/O, and PIGO. Graphs are arranged in size, with com-Friendster at 31 GB on disk and road-USA 940 MB.

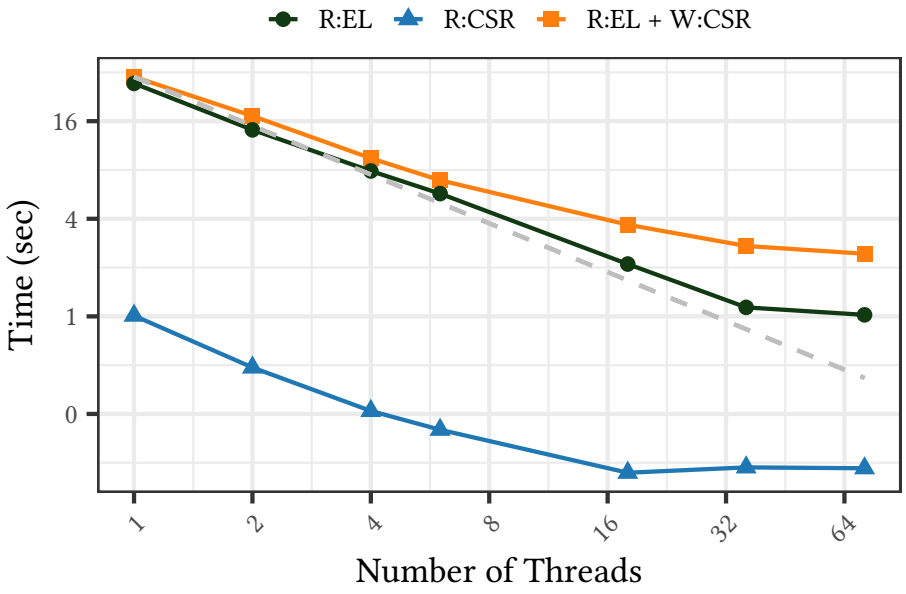


Figure 6.9: A scalability study showing the impact of increasing threads on UK-2002. The NUMA boundary is reached at 18 cores, after which scalability only continues to increase for ASCII processing and COO to CSR conversions.

in parallel, as evidenced by PIGO's overall performance gains. In Figure 6.8 we show the loading times for graphs both from `EL` and from preprocessed binary formats along with their conversion times. Galois uses `mmap` internally for binary files. Our parallel reading remains faster. Their ASCII file processing, however, is too slow to use in production. In fact, for Ligra and GAPbs, PIGO loading from `EL` and converting to CSR is faster than loading preprocessed binary files without PIGO.

We show the scalability in Figure 6.9. Up to the NUMA boundary scalability increases with all approaches. Overall, we have shown that PIGO provides significant performance improvements.

6.3 Summary

We tackle the long-standing belief that parallel I/O is not fruitful for loading sparse matrix and graph files. While there may be limited parallel improvements to cold cache raw binary reads over SATA, we show there is much to be gained with RAID controllers, NVMe SSDs, or a warm cache. We introduce a simple to use, header-only C++ library that enables both highly-tuned graph systems and small, one-off graph kernels to take advantage of parallel I/O. Our library is open source and it brings over 40× end-to-end performance improvements to state-of-the-art graph and sparse matrix systems.

CHAPTER 7

SCALING UP: MAINTAINING CORES IN PARALLEL

An important problem in graph analysis is finding locally dense regions in globally sparse graphs. In this work we consider the problem of finding k-cores [228, 183], which are maximal connected subgraphs with minimum degree at least k. This problem has seen significant attention due to its computational efficiency [183] and usefulness on a large number of problems [147, 9, 109, 251, 99, 85, 144]. We address computing k-core *values*, the largest k such that a vertex is in a k-core. Cores themselves can then be efficiently computed from the values [80].

Many practically important graphs today, from web data, social networks, and related fields, are both large and continuously changing. Finding dense regions as quickly as possible after a change is important, for example to quickly initiate a response to rapidly spreading false information about vaccines or to urgently address new pandemic super-spreading events. We focus on *maintaining* k-core values over a dynamic graph with batches containing both edge insertions and deletions. The maintained k-core values can then be directly queried. The goal of maintenance algorithms is to drive down the *latency* of a query, or the algorithm runtime for processing a single edge change. This typically comes at a cost of *throughput*, or the number of edge changes processed by the total runtime. A sequential, single-edge maintenance algorithm typically has both a low latency and throughput, whereas re-computing from scratch will have both a high latency and throughput.

There are two main approaches for maintaining cores values on dynamic graphs, traversal [225, 164] and order [274]. When bursts of large activity come in, they *cannot keep up* with the data stream. There has been a recent focus on parallel batch algorithms to address this problem [67, 236, 3, 59]. Such algorithms operate on a *batch* of edge changes at once, enabling more parallelism at the cost of latency for small batches. They provide a middle

ground between computing from scratch and single-edge maintenance algorithms. There are three known parallel batch algorithms for cores, all based on the idea of finding *independent* edges and processing them with traditional, sequential techniques [258, 131, 121]. Unfortunately, this approach does not show much scalability as additional processors are added (e.g., see Fig. 11 [131], Fig. 6 [121].) Given the increasing rate of data from social and web applications, there is a strong need for completely new approaches that can scale as the number of threads grow. We present two new algorithms to address this gap.

Our algorithms use the connection between h-indices [116] and k-cores, first identified by Lu et. al [174], which has been used as a local, distributed algorithm [196, 222] for computing from scratch. In this algorithm, each vertex has a local value which is initialized to inifinity. At each step (either synchronously or asynchronously), the algorithm computes the h-index of its neighbors' values. This process will converge within the number of degree levels of the graph[222]. The advantage of this process is that, after initialization, each vertex can operate independently. Building on this, we provide two scalable maintenance algorithms that maintain k-core values on batches of graph insertions and deletions. The first, mod, is based on modifying local values and then "continuing" convergence. The challenge with this approach is to increment local values as little as possible. The second, set, follows the h-index iterations, but keeps track of each edge insertion made to the graph. The increases happen locally, based on the given set of insertions and together with convergence. The mod algorithm provides consistent improvements over re-computing from scratch for large batches, and the set algorithm provides improvements for small batches.

In many cases, real-world data is naturally modeled as hypergraphs [233]. For example, consider purchasing relationships that consist of users and items. Hyperedges naturally model multiple users purchasing the same item. As another example, people may be vertices and hyperedges would indicate that they were close enough to each to spread diseases during a time period. Here, a hypergraph k-core would represent a group of people that are likely to internally spread disease. We want to ensure that our approaches both apply to

hypergraphs and have scalability when running on hypergraphs. For dynamic hypergraphs, there are two main models. One model treats each hyperedge as a single, immutable unit, and operates on a stream of hyperedge changes. This is the approach taken by [243]. However, this model cannot capture the dynamic nature of many existing hypergraphs. For example, consider a hypergraph consisting of users and topics that the user likes. In real networks, both new topics are created *and* users' preferences change. To model this behavior, each hyperedge itself can also have internal changes. We address this more general model.

We provide the following contributions.

- We provide two shared-memory parallel batch algorithms, mod and set, that maintain k-core decompositions using the connection between cores and h-indices on both graphs and hypergraphs
- We introduce strategies to handle parallelism in the more challenging case of changing hyperedge *pins*
- We demonstrate our algorithms' scalability empirically

The remainder of this chapter is structured as follows. In § 7.1 we provide background, including notation used, the problem we address, and related work. In § 7.2 we present the static algorithms for k-core decompositions. In § 7.3 we introduce our parallel batch algorithms. In § 7.4 we present our experiments and results, and finally in § 7.5 we conclude.

7.1 Background

7.1.1 Notation

Here we describe the notation used throughout the chapter. We are concerned with both graphs and hypergraphs. Let $G = (V, E)$ be a graph, where V is a set of vertices and E a set of edges. We focus on simple, undirected graphs, and so each edge is a set containing

two distinct vertices. A hypergraph $H = (V, E)$ is a generalization of graphs, where V is a set of vertices and E is a set of hyperedges. In hypergraphs, a hyperedge is a subset of vertices, $e \in E$ such that $e \subseteq V$. This means that a hyperedge may contain one or more participating vertices, compared with exactly two as in a graph. We call the relationship between a vertex and a hyperedge a *pin*.

The set of neighbors of a vertex in both cases is given by $\Gamma(v) = \{u \in V : \exists e \in E, \{u, v\} \subseteq e\}$. The *degree* of a vertex is the number of neighbors it has, $\deg(v) = |\Gamma(v)|$. The maximum degree in a graph or hypergraph is $\Delta(G)$ ($\Delta(H)$, resp.). Induced subgraphs in hypergraphs cannot split hyperedges, and so every pin in a hyperedge remains. So, when an induced subgraph is taken (for example, in a k-core), if *any* vertex in a hyperedge has a degree less than k then the hyperedge is effectively "peeled" from the hypergraph.

7.1.2 Dynamic Hypergraphs

For hypergraphs, the stream can be either at the level of *hyperedge changes* or *pin changes*. In either case, the notion of a k-core still requires inducing a full hyperedge, separating this problem from that of bipartite cores [171]. While [243] addresses hypergraph streams with hyperedge changes, we address the more general model of *pin changes*. It is straightforward to simulate hyperedge changes by setting batch boundaries at full hyperedges.

7.1.3 Hypergraph k-Cores to Address Pandemics

Computing hypergraph k-cores may be beneficial for identifying groups of individuals to monitor or address for spreading of diseases, for example to assist in addressing the COVID-19 pandemic. Consider Figure 7.1. Here, a hyperedge (a rectangle) is created between any individuals that have close contact within a time period. In the figure, person A is in a meeting with person B and E, and so they have close contact and the first line is a hyperedge connecting them. We call such a hypergraph a *co-occurrence* hypergraph. A k-core in this hypergraph could then identify individuals with significant close contact with

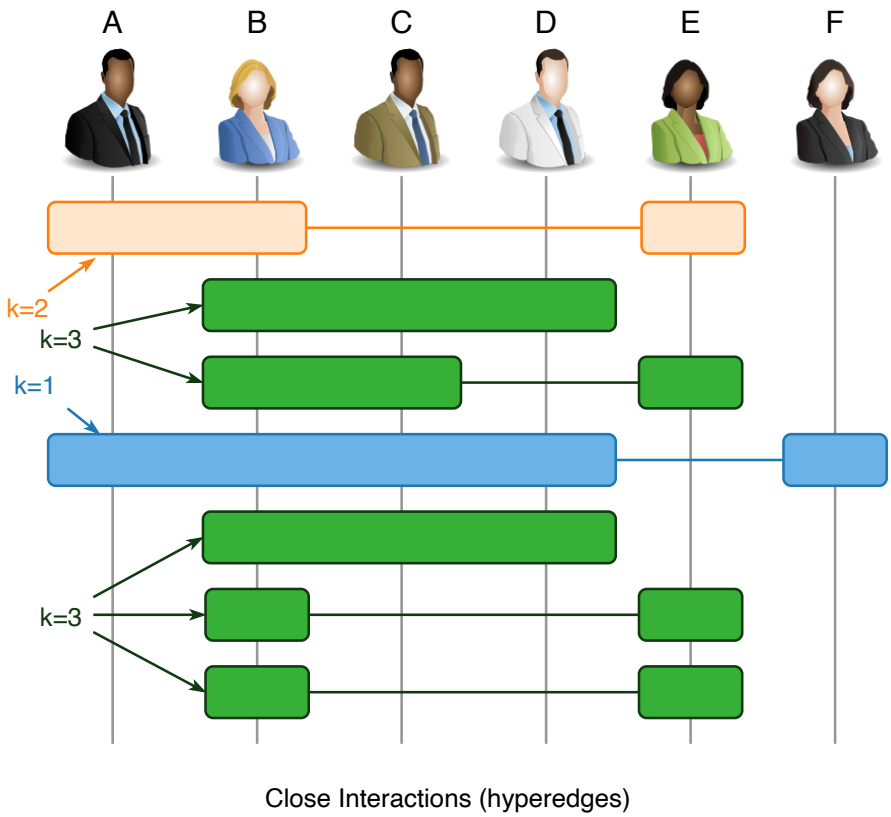


Figure 7.1: Co-occurrence hypergraphs may be useful for identifying groups susceptible to pandemics. Boxes are hyperedges. Dark green indicates all members have $\kappa = 3$, orange indicates $\kappa = 2$, and blue indicates $\kappa = 1$. A graph representation gives F a high degree and a high core value, even though it likely has low exposure. In a co-occurrence hypergraph, F would have a core value of 1 whereas B, C, D, and E would have a value of 3, capturing that they have close, on-going, and intimate interactions.

others, in a way that captures deeper relationships than simply looking at a graph perspective. For example, for person F, a graph representation would show them connected with every other individual, with both a high degree and a high core value. However, they may have been present only at one meeting, and so would be less likely to catch the disease.

The hypergraph k-core instead would place B, C, D, and E in a 3-core together, as they are each part of 3 hyperedges where all members are in a 3 hyperedge as well.

Algorithm 7.1: Asynchronous local algorithm to compute κ [174].

```
   Data: graph G = (V, E)
1  ∀v ∈ V, τ[v] ← deg(v)
2  repeat
3  |  for v ∈ V do in parallel
4  |  |  L ← ⟨⟩
5  |  |  for w ∈ Γ[v] do
6  |  |  |  L ← L.extend(τ[w])
7  |  |  τ[v] ← H-INDEX(L)
8  until τ no longer changes, converging to κ
9  return τ
```

7.2 Static h-index Algorithms

In this section we describe h-index based approaches to compute cores, which are the foundation of our maintenance algorithms.

Definition 7.1. *Let $S = \langle s_1, s_2, \ldots, s_n \rangle$ be a tuple of values, with $s_i \in \mathbb{Z}$ for $i \in [1, n]$. The h-index of S is the largest value h such that $s_i \geq h$ for $i \in H$, where $|H| = h$ and $H \subseteq [1, n]$.*

In this section we describe the prior foundational algorithm, highlight the challenges in converting them to dynamic *maintenance* algorithms, and describe our final algorithms.

7.2.1 h-index Coreness Computation

To understand our algorithms, it is first useful to understand the asynchronous local h-index algorithm proposed by [174]. This algorithm is re-presented here in Algorithm 7.1 for completeness. The local variable τ is initialized to the degree of each vertex in H. Then, each vertex is iteratively processed and τ is updated based on the h-index of neighbor's τ values. When no further changes occur $\tau = \kappa$. This differs from the synchronous version presented in Algorithm 2.2 as there is no explicit version dependency on τ.

This was a breakthrough for computing k-cores as it is well suited for parallelization: each vertex can update its own local values, given access to its neighbors' local values, in parallel. In the synchronous version each vertex considers its neighbor's values from the

previous time step. In the asynchronous version, each vertex takes the latest available value for each neighbor.

7.2.2 Key Problem: How To Reinitialize

Consider Algorithm 7.1. There are many different possible initializations for τ. In fact, τ can be initialized to any value equal to or larger than κ—the degree is chosen simply because it is an upper bound on κ. It would be possible to use ∞, or $\kappa[v] + 1$, as shown by the convergence of the asynchronous version [174]. If τ is initialized to ∞, then after only one iteration τ will recover the degree initialization. The constraint is that τ cannot be initialized *too low*, that is, if some vertex d, $\tau[d]$ is initialized below $\kappa[d]$, τ may fail to convergence to κ.

It may seem like simply re-using the prior output, incrementing the edge that changed, and continuing the computation will work. Unfortunately, this is not the case.

Lemma 7.1. *If a τ value is below κ, then Algorithm 7.1 may never converge to κ.*

Proof. Consider P_n, a path of length n. Note that $\forall v \in P_n$, $\kappa[v] = 1$. Let the endpoints be v_1 and v_n, such that $\deg(v_1) = \deg(v_n) = 1$. Suppose $\{v_1, v_n\}$ is an inserted edge. Then, suppose that *any two consecutive* vertices v_j, v_k have $\tau[v_j] = \tau[v_k] = 1$. Now, running Algorithm 7.1 to convergence will result in $\kappa[v_i] = 1 \forall i \in [1, n]$, which is incorrect. □

As tempting as it is, given Lemma 7.1 a simple memoization algorithm *will not work.*

Furthermore, if only part of the hypergraph is initialized above κ, then the vertices already at κ do not need to re-compute each iteration allowing the problem to remain local to part of the graph.

The key problem is that we need to know the smallest set to increment, and increment those vertices, before we can run to convergence.

Building on this observation we present two dynamic algorithms along with several variants. In the first algorithm, we initialize τ values once for each batch, trying to increase

Algorithm 7.2: hhcLocal, extending [174] to hypergraphs.

Data: hypergraph $H = (V, E)$
Input: optional τ initialization, frontier A

if τ *is not given* **then** $\forall v \in V, \tau[v] \leftarrow \deg(v)$
if A *is not given* **then** $A \leftarrow V$
repeat
  for $v \in A$ **do in parallel**
    $A' \leftarrow \emptyset$
    $L \leftarrow \langle\rangle$
    for $e \in E : v \in e$ **do**
      $L \leftarrow L.\mathsf{extend}\left(\min_{w \in e, w \neq v} \tau[w]\right)$
    $\tau[v] \leftarrow \text{H-INDEX}(L)$
    if $\tau[v]$ *changed* **then** $A' \leftarrow A' \cup \{v\} \cup \Gamma(v)$
  $A \leftarrow A'$
until $A = \emptyset$
return τ

τ as little as possible while ensuring convergence. We keep track of which vertices in H are already at convergence and do not perform any computation on them, allowing for batches to run in $o(|H|)$. In the second, we combine initialization and convergence, allowing for initialization to spread concurrently with convergence. We keep track of changes to H and propagate them through the graph, increasing τ values for neighbors that have not seen the change yet and would be affected by it.

7.2.3 Extension To Hypergraphs

We extend Algorithm 7.1 to hypergraphs. Our extension is shown in Algorithm 7.2. In particular, we build the neighbor list L using the *minimum value* after excluding the source vertex. This allows for coreness values to remain correct, as any vertex with too low of a coreness value will cause the entire hyperedge to stop contributing.

Theorem 7.1. *Algorithm 7.2 will correctly return the coreness values κ.*

Proof. This proof follows Thm. 1 [174], using induction to bound the h-index sequence from above and below, resulting in exactly the coreness. □

Given the similarity between k-cores on graphs and hypergraphs, for the remainder of this chapter we stay in the hypergraph setting. The special case, when each hyperedge has exactly two endpoints, applies in all of the results and is easy to handle as a special case for an implementation.

7.3 h-Index Based Core Maintenance

We now present two dynamic algorithms, both of which build on the local h-index algorithm presented in [174]: the first involves incrementing τ across the graph, attempting to increment as few times as possible. Convergence then occurs similarly to in Algorithm 7.1. The second involves combining initialization and convergence. We keep track of each change to H and run a modification of the h-index algorithm. As the τ computation iterates, updates are propagated outwards, causing vertices to increment their own τ as appropriate. We refer to this algorithm as set, with setmb additionally optimized with mini-batches.

These two algorithms come with different tradeoffs. The re-initialization is useful to provide consistent latencies lower than a static recomputation, but on many graphs fails to capture really low latencies. The combined initialization and convergence provides a different advantage: it can have significant latency improvements, reaching over $10^4\times$ static computation on real-world graph instances, but with high variability: with some batches there are far more iterations as increments propagate, and there is a computational overhead for checking whether an update has been processed.

Our algorithms are presented with *callback* functions, which are designed to be run when an vertex is inserted into or removed from a hyperedge. The change value c is the direction, indicating either an insertion (+) or deletion (−).

Algorithm 7.3: A simple variant of mod that operates only on a single *hyperedge* change and maintains κ. hhcLocal extends Algorithm 7.2 with active vertices that, on a local change, make neighbors active and otherwise go dormant.

Data: hypergraph $H = (V, E)$, local values τ
Input: single hyperedge change $y = \{d_1, \ldots, d_s\}$, c

▷ update the hypergraph

for $d_i \in y$ **do**
  if $c = +$ **then** $H[y, d_i] \leftarrow 1$
  else $H[y, d_i] \leftarrow 0$

▷ maps are zero-initialized

$R \leftarrow \{\}$ ▷ map τ values to num. resolved ins.
$D \leftarrow \{\}$ ▷ map τ values to num. of deletions

▷ find a vertex with min τ value

$d_m \leftarrow \arg\min_{d_i \in y} \tau[d_i]$
if $c = +$ **then** $R[\tau[d_m]] \leftarrow 1$
else $D[\tau[d_m]] \leftarrow 1$
$A \leftarrow \emptyset$ ▷ active vertices to process
for $d \in V$ **do in parallel**
  ▷ apply the resolved count
  $\tau[d] \leftarrow \tau[d] + R[\tau[d]]$
  ▷ mark changed vertices as active
  if $R[\tau[d]] \geq 0$ *or* $D[\tau[d]] > 0$ **then**
    $A \leftarrow A \cup \{d\}$
hhcLocal(A, τ), using vertices A and initialized to τ

7.3.1 Re-initialization Based Algorithms

In this section we describe our algorithms that are based on initializing τ and then running Algorithm 7.2, which result in fewer iterations than static computation in many instances. This problem was shown to be unbounded in the locally persistent model [273], and so we cannot expect to do better than re-running from scratch in all cases. This result also means that we need to ensure our worst-case complexity matches computing from scratch, but we cannot expect to have a better complexity.

Differing significantly from state-of-the-art core and truss maintenance algorithms [273, 274], we do not maintain an order of the vertices. Instead, we make larger increments than necessary and let the parallel local h-index approach resolve them.

We begin by introducing a simple, non-batch variant of our algorithm, presented in Algorithm 7.3. Our batch approach naturally extends from there to the full mod algorithm, and is shown in Algorithm 7.4.

We know that for a single edge insertion in H, the κ value will only change for the involved minimum κ valued vertex (by [225]). However, we will be performing multiple updates in a batch. This means that the τ values are, at this point in time, potentially *smaller* than κ. So, we cannot identify which vertex has the minimum κ and instead increment all κ values that participate in the modified hyperedge. In lines 9-16 we deal with the situation where the subcore, or part of it, has *moved* after some other insertion or deletion. We conservatively increment in as many potential parallel cases as possible, such as when the subcore is broken, part of it merged with smaller subcores, and more.

It is possible to limit checking each vertex in line 18 by using reverse maps, however in practice, the overhead of this reverse map offsets potential gains.

If a κ value changes for a vertex, two properties must hold: that vertex must have a specific starting κ value and they must be connected to the hyperedge change (again by [225]). The mod algorithm exclusively uses the κ *value* and ignores the connectivity.

Additionally, we perform an important optimization: the minimum values on hyper-

Algorithm 7.4: The mod algorithm.

Data: hypergraph $H = (V, E)$, local values τ

▷ Insertion callback

Function `f-mod`(e_a, v_b, c)**:**
  if $\exists v_i \in e_a$ *s.t.* $\tau[v_b] > \tau[v_i]$ **then return**
  if $c = +$ **then** $I[\tau[v_b]] \leftarrow I[\tau[v_b]] + 1$
  else $D[\tau[v_b]] \leftarrow D[\tau[v_b]] + 1$

Input: batch edge set B

$I \leftarrow \{\}$ ▷ map τ values to num. of insertions
$D \leftarrow \{\}$ ▷ map τ values to num. of deletions
$R \leftarrow \{\}$ ▷ map τ values to num. resolved ins.
MaintainH(`f-mod`, B)

▷ increment as many possible subcore levels and values that could arise from concurrent execution

for $k \in$ keys(I) **do in parallel**
  ▷ increment as if some subcore at $\kappa = k$ decreased and merged with another
  for $t = k - D[k]$ *to* $k - 1$ **do**
    $R[t] \leftarrow R[t] + I[k]$
    $R[k] \leftarrow R[k] + I[t]$
  $R[k] \leftarrow I[k]$ ▷ increment if stayed at level
  ▷ increment as if some subcore at $\kappa = k$ increased and merged with another
  for $t = k + 1$ *to* $k + I[k]$ **do**
    $R[t] \leftarrow R[t] + k + I[k] - t$
    $R[k] \leftarrow R[k] + I[t]$
$A \leftarrow \emptyset$ ▷ active vertices to process
for $d \in V$ **do in parallel**
  ▷ apply the resolved counts
  $\tau[d] \leftarrow \tau[d] + R[\tau[d]]$
  ▷ mark changed vertices as active
  if $R[\tau[d]] \geq 0$ *or* $D[\tau[d]] > 0$ **then**
    $A \leftarrow A \cup \{d\}$
hhcLocal(A, τ), using vertices A and initialized to τ

edges are cached. It is possible to only store a single minimum, as this will not have a negative impact on the convergence or correctness.

7.3.2 Processing in Parallel with Pin Changes

The problem becomes more complicated for re-initialization algorithms as we deal with a stream of *pin changes* instead of hyperedge changes. For each pin deletion, this can result in both *an increase* and *a decrease* in κ values for different nodes. The hyperedge that *loses* the pin may in fact *gain* a new κ value for all other vertices. This will happen if the pin is exactly the lowest valued vertex in the hyperedge. Additionally, the vertex that the pin connects to may drop its k-core value.

Similarly, with an insertion, this can result in both an *increase* in the core value for the vertex with the pin and a *decrease* in the core value for every other vertex in the hyperedge.

This complicates the decision for resolving increments, and increases the number of increments and decrements that have to happen. This results in significantly more book-keeping for the process in lines 9-16. The full, parallel process in the implemented system can be seen in Figure 7.2. Here, given a change to the hypergraph, we need to keep track of whether the change can be processed immediately or needs to be resolved later. This process will iterate until all changes have been resolved. The need for this work can be seen in Figure 7.3.

Concretely, we break the problem down into four cases. Here, we present the cases dealing with deletions. For insertions, the deletions and insertion changes are swapped.

- Case 1: the hyperedge no longer exists. Find the minimum range within the previous hyperedge vertices' ranges and decrement this range.

- Case 2: the minimum range of the deleted pins can be smaller than the existing minimum range. Here, decrement everything within the minimum range of the deleted pins, and increment everything within the minimum range of the existing hyperedge.

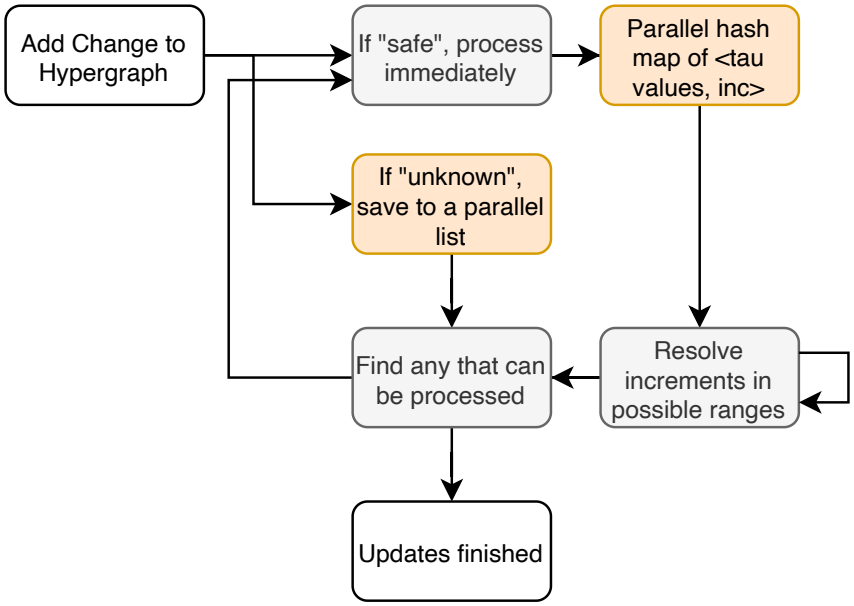


Figure 7.2: The process of performing increments based on a hyperedge change. Safe means the change can be concretely resolved, that is the *possible positions* of the pin in the hyperedge *range* is known.

- Case 3: no deleted pins had vertices within the minimum range. This is marked *unknown*, and needs to be revisited in the next loop if anything else can change.
- Case 4: the deleted pins minimum range may overlap the existing minimum range. Decrement everything in the middle range.

Note that everywhere we use *ranges* instead of concrete κ values, as they will be processed in a loop until convergence. During the processing, we make sure not to re-do work. Even though ranges will grow during iterations, the new increments and decrements, and changed cases, should not re-increment prior values.

7.3.3 Mixing Initialization and Convergence

In this section we describe our set algorithm, which mixes initialization of τ and convergence concurrently. This algorithm deviates internally from Algorithm 7.2. Each hyperedge change is recorded and its history is remembered for the course of the batch. During each h-index computation, the neighbors of a vertex are considered but instead of reading τ

Algorithm 7.5: The set algorithm, which mixes incrementing τ and converging τ. The id function resets each batch and increments on distinct e_a inputs.

Data: hypergraph $H = (V, E)$, local values τ

▷ Insertion callback

Function `f-set` (e_a, v_b, c)**:**
  $A[v_b] \leftarrow 2$ ▷ maximum time-to-live
  if $c =$ + **then** $U[v_b] \leftarrow U[v_b] \cup \{\mathsf{id}(e_a)\}$

Input: batch edge set B

$A, U, P \leftarrow \{\}, \{\}, \{\}$

MaintainH(`f-set`, B)

repeat
  $c \leftarrow$ false
  for $x \in V$ **do in parallel**
    if $A[x] = 0$ **then** continue
    $U_x \leftarrow U[x]$ ▷ $U[x]$ may change
    $L \leftarrow \emptyset$ ▷ list of neighbor values
    for $e \in E : x \in e$ **do**
      $m \leftarrow \infty$
      for $n \in e : n \neq x$ **do**
        ▷ Consider un- or processed hypergraph changes for n
        $t \leftarrow \tau[n] + |U[n] \cup (U_x \setminus P[n])|$
        if $t < m$ **then** $m \leftarrow t$
      $L \leftarrow L \cup m$
    $\tau' \leftarrow$ H-INDEX(L)
    ▷ Determine if our τ changed
    if $\tau' \neq \tau[x]$ **then**
      ▷ Update the neighbors
      for $n \in e$ *such that* $x \in e$ *and* $n \neq x$ **do**
        $U[n] \leftarrow U[n] \cup (U_x \setminus P[n])$
        $A[n] \leftarrow 2$
      $\tau[x] \leftarrow \tau'$
      $A[x] \leftarrow 2$
    else
      $A[x] \leftarrow A[x] - 1$
    $P[x] \leftarrow P[x] \cup U_x$
    $U[x] \leftarrow U[x] \setminus U_x$
    $c \leftarrow$ true
until $c =$ *false*

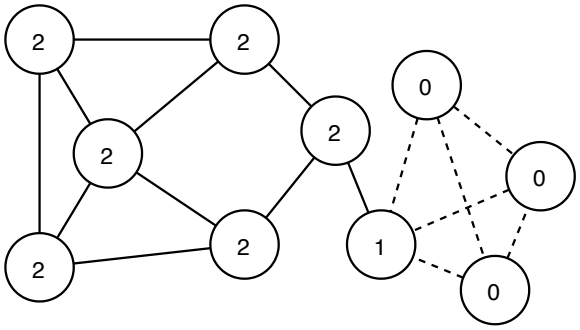


Figure 7.3: A notional example showing why increments need to be sufficiently high. The dotted lines are new edges and the solid lines are existing edges. Even though edges are only added to the $\kappa = 1$ vertex, after the batch is processed all vertices need to increase to $\kappa = 3$.

directly, τ is read and each potential graph change is applied. If this will result in a change to that vertex, then the change is propagated. Otherwise, the change stops.

This allows for a small part of the graph to be visited, but in the worst case will result in additional iterations and additional work, as each change may slowly propagate to the whole graph *before* regular τ convergence, increasing the number of iterations by the diameter of the graph. We present this approach in Algorithm 7.5. In the callback, each vertex is marked as having the modification "unprocessed." Then, lines 10-28 contain the mixed convergence with increments. First each neighbor is considered. Instead of setting the new τ value based on the h-index of the neighbor's τ values, it sets τ based on the h-index the neighbor's currently unprocessed modifications, plus the modifications neither the vertex nor the neighbor has processed. Differing from the re-initialization algorithms, vertices stay active for one extra iteration after convergence. This allows for convergence to occur when $U[x]$ is updated while x is currently processing and covers cases where an update propagation changes from incrementing to iteratively converging.

While the updates are propagating the memory may continue to grow across the hypergraph, as more U and P entries are being set. There are a variety of ways to realize this in implementation, some of which come with more expensive memory requirements. We considered using boolean vectors, dynamic bit vectors, and fixed-size pre-allocated bit

vectors coupled with mini-batches. Our experiments are all performed with mini-batches (setmb), with batch sizes of 64. Mini-batches stopped iterating when P became empty for all vertices with a final batch iteration to converge τ.

The correctness of this algorithm comes from the observation that if the frontier of an update will not cause any further increase in τ values, then it is not necessary to further propagate the update.

7.4 Experiments and Results

In this section we perform experiments to empirically demonstrate the scalability of our two algorithms, mod and setmb, as graphs become bursty.

We implemented our algorithms in C++17 and compiled with GCC 10.2.0 and `-O3`. We use Intel TBB to provide parallel hash maps to store the graph (with the edge lists stored as vectors) and both TBB and OpenMP to parallelize execution. We ran on Intel Xeon E5-2683 v4 CPUs with 512 GB RAM and dual sockets. We use `numactl` to control the number of threads. The correctness of graph and hypergraph results were checked by comparing with Ligra [233].

Unfortunately we were not able to compare against existing alternative parallel implementations, as they are not publicly available [258, 131, 121]. We note that against the *reported runtimes*, we are around 4× faster, however we are using different hardware and systems. We stress that none of the prior systems have reported scalability as cores are added.

7.4.1 Datasets

We choose a variety of graphs from domains representing social networks, citation networks, and web data. The graphs are shown in Table 7.1 and the hypergraphs are shown in Table 7.2. The graph datasets were download from SNAP [160] and the hypergraphs from KONECT [151]. As these graphs are not ordered temporally and do not have deletions, we

Table 7.1: Graphs used for our experiments.

Name	Vertices	Edges
OrkutLinks	3.07 M	240 M
LiveJ	3.99 M	37.4 M
Pokec	1.63 M	22.3 M
Patents	3.77 M	16.5 M
DBLP	1.82 M	8.34 M
WikiTalk	2.39 M	4.66 M
Google	0.88 M	4.32 M
YouTube	3.22 M	9.38 M

Table 7.2: Hypergraphs used for our experiments.

Name	Vertices	Hyperedges	Pins
OrkutGroup	2.8 M	8.7 M	327 M
WebTrackers	27 M	13 M	141 M
LiveJGroup	3.2 M	7.5 M	11.M

simulated edge insertions and deletions as follows. First, we select the batch size number of edges and remove them from the graph. In the hypergraph case, we selected the number of pin changes. We then insert them back again, and time both the removal and insert. To test mixed insertion and removal times, we set our removal and insert size to be $3/2$ the full batch size. Then, for every removal and insert test, we have one batch with only removals, one mixed batch, and one batch with only insertions.

In each experiment, the batches were removed and then re-inserted 50 times. The error bars in all of the plots show one standard deviation from the mean. We choose batch sizes based on expected real-world ranges for each algorithm. We do not show the results for setmb for hypergraphs, as it will require caching values on hyperedges to be competitive against mod.

7.4.2 Insertion Scalability

First, we measure the scalability for handling insertions for both mod and setmb. The results can be seen in Figure 7.4–7.5. In both cases, as the cores increase the total run-

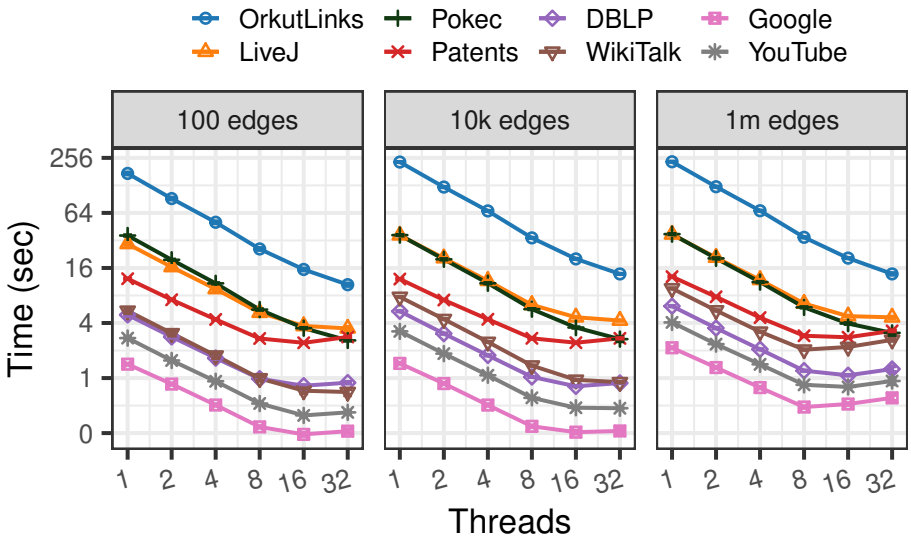


Figure 7.4: The mod algorithm's scalability for processing edge insertions at different batch sizes. Insertion-only edge batches with mod.

time decreases. The total runtime for small batches with mod is only slightly less than large batches, showing that as the batch size increases the algorithm similarly scales well. Choosing to run setmb for very small batches and mod for larger batches would be effective for a wide range of insertions. Additionally, setmb has a very high variance on the larger graphs. While it provides the smallest runtimes on small batches, it also has high outliers that significantly increase the average. Future work can address reducing the variance and the maximum cost.

For some graphs, moving from 16 to 32 threads comes with a slight decrease in performance.

In Figure 7.6 we show the scalability for running mod on the hypergraphs. For WebTrackers, the performance decreases in all cases after 8 threads, however for OrkutGroup and LiveJGroup the performance continues to decrease after the NUMA boundary. With up to 8 threads on those graphs the scalability is close to linear.

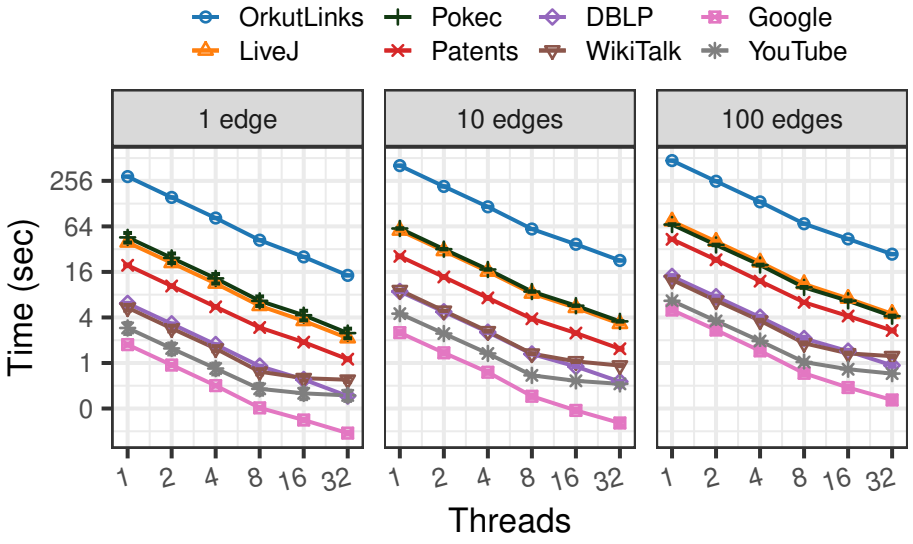


Figure 7.5: Insertion-only edge batches with setmb.

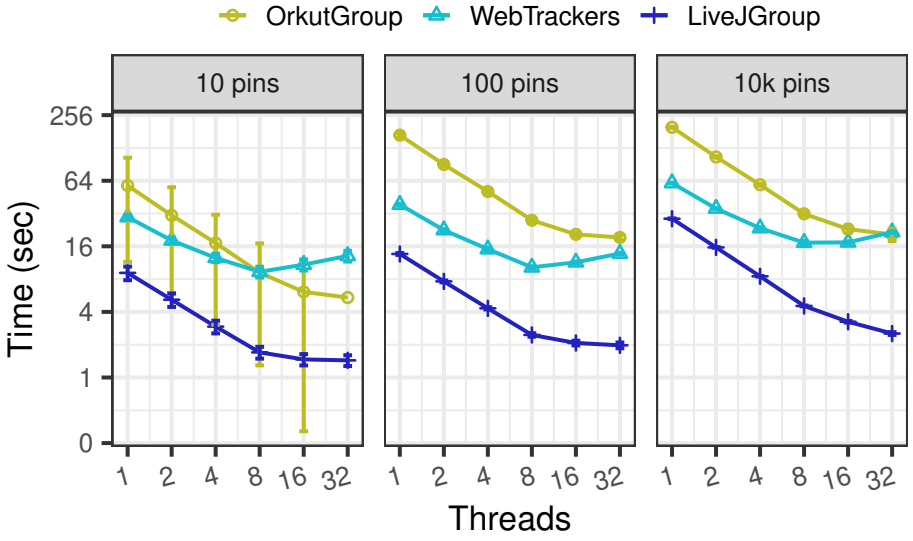


Figure 7.6: Insertion-only pin batches with mod.

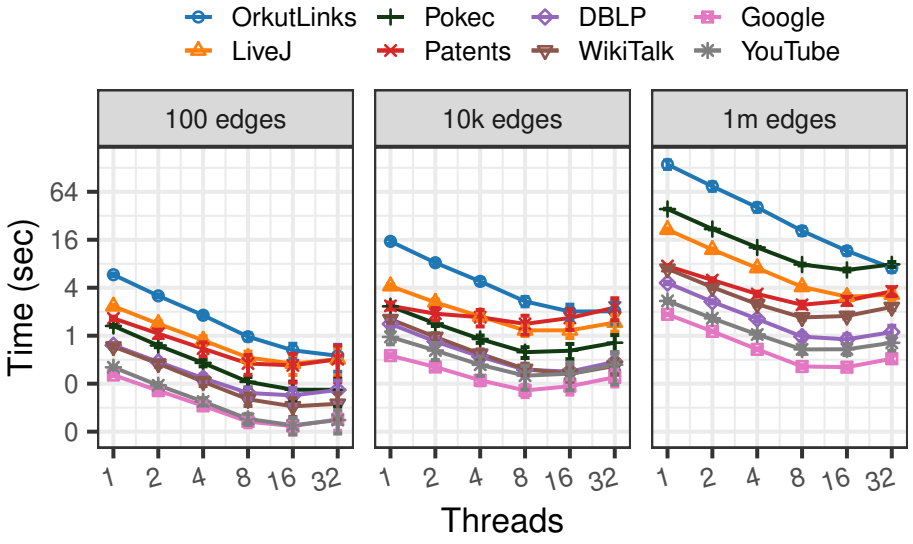


Figure 7.7: Deletion-only edge batches with mod.

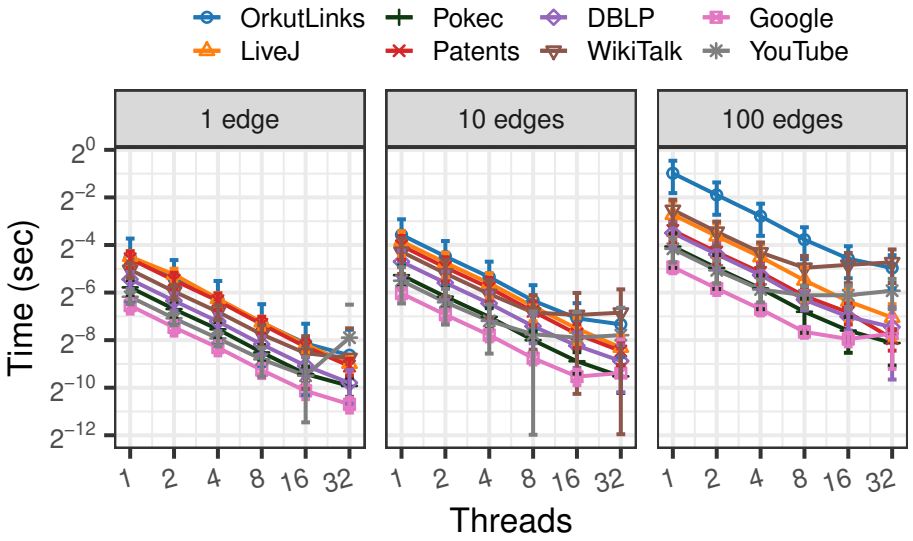


Figure 7.8: Deletion-only edge batches with setmb.

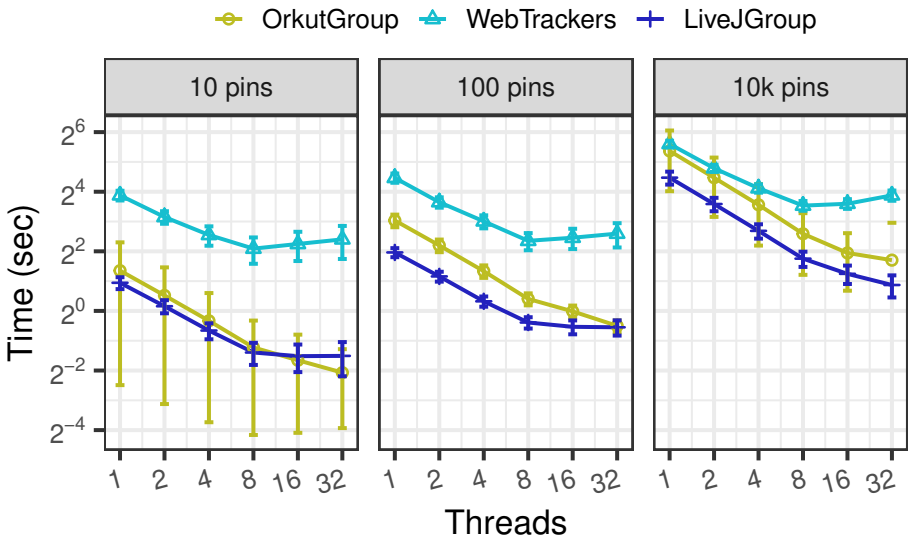


Figure 7.9: Deletion-only pin batches with mod.

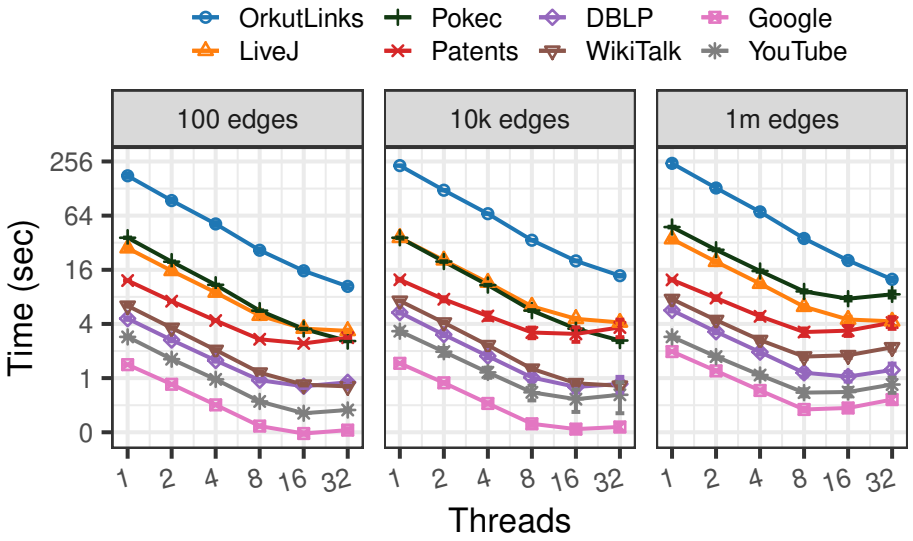


Figure 7.10: Mixed batches with mod.

7.4.3 Deletion Scalability

We measure the scalability for performing just edge deletions with both mod and setmb. The results can be seen in Figures 7.7–7.9. For both mod and setmb, the performance tends to decrease as the batch sizes get larger and increase as the number of threads increasing, showing that this approach similarly scales on deletions. For setmb, even with large batches the latency for deletions is low.

When deleting pins in hypergraphs, the variance can become large. For example, see OrkutGroup with 10k pins. Both the insertion and deletion variance for a small number of pin changes is high.

7.4.4 Mixed Insertions and Deletions

One advantage of these algorithms is that they do not require pre-processing on the stream to separate batches into deletions and insertions. Instead, insertions and deletions can be handled concurrently. Our results show similar scalability for mixed batches as with insertions. For example, in Figure 7.10 we show the mixed batches for mod. Note the similarity to Figure 7.4.

7.5 Summary

In this chapter, we presented two scalable, parallel batch k-core maintenance algorithms that operate on fully dynamic graph and hypergraph streams. These algorithms differ from prior approaches in that they build on the connection between h-indices and k-cores. We address two models for dynamic hypergraphs, one with hyperedge changes and one with pin changes. Our implementations empirically scale well on shared-memory systems, exceeding the scaling performance of prior algorithms.

Future work includes combining the two approaches into a hybrid approach that can provide both low latencies for small batches but addresses high variance, introducing ap-

proximate results during very high batch rates, and implementing these algorithms in distributed systems to further explore scalability.

CHAPTER 8

DISTRIBUTED FAST h-INDEX COMPUTATION

Computing h-indices on large lists is an increasingly important kernel. Prior sequential algorithms, which are based on bucket sorting, suffer from large communication costs when directly parallelized. Building on fast selection algorithms, in this chapter we introduce DHIndex, a new parallel algorithm that uses divide-and-conquer with a pivot to compute the h-index. Using the median as a pivot, DHIndex has a computation complexity of $O(n/p)$ and a communication complexity of $O(\log p \log n)$, improving on the $O(n/p + \log p)$ communication cost from a direct parallelization of sequential algorithms, where n is the size of the list and p the number of processors. Similar to selection algorithms, the pivot can be chosen randomly to achieve the same complexities in expected time. Using a random pivot, DHIndex computes h-indices on over 3 trillion integers across 6784 cores in under 10 seconds. Even sequentially, DHIndex provides a speedup over prior state-of-the-art approaches on large inputs.

This is a crucial kernel for implementing core and nuclei computation and maintenance as we scale out, as it is the main computational problem for all h-index based approaches.

8.1 Introduction

The h-index [116] was proposed as a way to evaluate the influence of a researcher based on their publication record, defined as the largest number h of papers with at least h citations per paper. This index has become widely used for evaluating researchers, institutions, and journals [7], and has had many extensions and derivatives [35]. Beyond bibliometrics, the h-index is used to rank entities in domains such as Internet media [120] and sports [22]. Furthermore, the h-index, when iteratively applied, results in the degeneracy of a graph [174], meaning k-cores can be computed and even maintained in parallel using

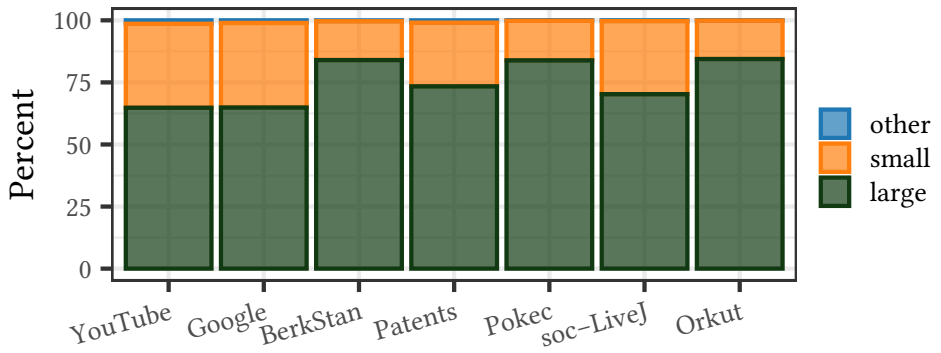


Figure 8.1: Time spent computing h-indices for large and small vertices and other computation for a k-core decomposition. A large vertex has a degree of at least 5 million, after which running DHIndex on a single thread is faster than prior approaches.

h-index computations [196, 222, 91]. Yet other promising uses for h-indices are emerging [70], including solving minimum problems in $L^2[0,T]$ [69] and geometric relations with Lorenz curves [68].

In this chapter, we address a largely unstudied yet increasingly pressing problem: how can we can compute the h-index quickly on *large lists*, with millions or more elements? When only considering author bibliometrics, problems at this scale are rare. However, as the h-index becomes a key kernel in other domains, such as k-cores and non-bibliometric indices, input datasets contain millions or billions of elements. Prior sequential algorithms are fast and take only two or three linear passes over the input. However, to the best of our knowledge, no prior parallel approach has been proposed. While the h-index problem is related to selection (finding the k-th smallest element in a list), fast selection algorithms [30] do not directly apply. We develop a new algorithm, DHIndex, that computes the h-index of a list work-efficiently in parallel. On shared-memory systems DHIndex provides speedups on large lists and on distributed-memory systems DHIndex enables previously out-of-reach data sizes.

We are motivated in particular by computing k-cores. Figure 8.1 shows the time spent computing k-cores by [222] in parallel on a variety of SNAP graphs [160]. Large vertices are defined as those with a degree of at least 5 million. We measure the time spent computing h-indices on neighbor lists for large vertices, small vertices, and any other computation

computation for h-indices. On average, over 75% of the time is spent on large vertices. A parallel h-index algorithm could enable improved load balancing and reduce the large vertex computation time. Additionally, in vertex-centric distributed models, parallel h-index computation is necessary when vertex neighbor lists are spread over multiple machines.

DHIndex is a recursive algorithm and works in a similar way to fast selection algorithms [30]. It takes multiple linear passes over the input, which can be done independently and in parallel. Each pass chooses a pivot element and then counts elements below, at, and above the pivot. The counts are reduced and used to determine if the h-index is the pivot, above it, or below it. If above or below, the input is filtered and DHIndex is recursively called.

8.2 Background and Prior Approaches

Let $S = \{s_1, \ldots, s_n\} \in \mathbb{N}$ be a multiset of integers. The h-index is the largest index i such that $|\{s \in S : s \geq i\}| \geq i$.

8.2.1 Bucket-Based Computation

An intuitive and natural way to compute the h-index is by sorting the list and then iterating through it in reverse, stopping when the position iterated through exceeds the value. Implemented directly, this would lead to $O(n \log n)$ runtime. An important observation is that the h-index cannot be larger than n, and so values above n can be replaced by n. By adjusting the sorting method to use a bucket sort with n elements, the h-index can be computed in two or three linear passes, depending on whether the maximum is computed. This approach is shown in Algorithm 8.1.

Correctness After line 4, A contains the counts of elements. t counts the number of elements visited, and the algorithm terminates on i if $t \geq i$. Following elements will be $< i$, and so i is the h-index.

Algorithm 8.1: Bucket-based fast h-index computation

Input: S

$M \leftarrow \min\{\max\{s \in S\}, n\}$; $A[M] \leftarrow [0, \ldots, 0]$; $t \leftarrow 0$
for $s \in S$ **do**
  $v \leftarrow \min\{s, M\}$
  $A[v] \leftarrow A[v] + 1$
for $i \in \{M, \ldots, 1\}$ **do**
  $t \leftarrow t + A[i]$
  if $t \geq i$ **then return** i
return 0

Complexity M can be at most n, bounding the for loops and max. As such, the runtime and space requirements are both $O(n)$.

Parallelization With large datasets, parallelization becomes important. We consider a natural extension of this algorithm to the parallel case. M is computed with a reduction. The array A is distributed across the ranks and an all-to-all communication moves counts to the corresponding location in A. Then, the final for loop is computed with a prefix sum. This results in $O(n/p)$ computation complexity and $O(n/p + \log p)$ communication complexity.

8.3 DHIndex

We first introduce a sequential algorithm, running in $O(n)$, and then describe its parallelization and resulting complexities.

8.3.1 Overview

This algorithm is recursive and reduces the amount of data processed by a constant factor in each recursion step. First, a pivot is selected. Then, each element in the list is compared against the pivot and the number of elements larger, smaller, or identical to the pivot are counted. Based on these counts, it is possible to determine whether the h-index is higher, lower, or equal to the pivot element. The algorithm then appropriately recurses on the

reduced multiset.

Lemma 8.1. *Let S be a multiset of integers and $p \in S$ a pivot element. Denote $n = |S|$, $a = |\{s \in S : s > p\}|$, $b = |\{s \in S : s < p\}|$, and $e = |\{s \in S : s = p\}|$. Let the h-index of S be h. Then,*

- *If $a > p$, then $h > p$*
- *If $a + e < p$, then $h < p$*
- *Else, $h = p$*

Proof. Suppose $a > p$. There are at least $p + 1$ elements valued at least $p + 1$, and so $p + 1$ is a lower bound on h. Hence, $h > p$.

Suppose $a + e < p$. By definition, p is not an h-index itself and is an upper bound. Hence, $h < p$.

Suppose otherwise $a \leq p$ and $a + e \geq p$. We know that there are at least p elements greater than or equal to p, and so p can be an h-index if it is the largest integer p. Suppose there was a larger integer, $p + 1$. This is not an h-index, by assumption, as there are at most p integers valued $p + 1$ or higher. Hence, $p = h$. □

Using Lemma 8.1, we can move up or down. The final point of the algorithm covers how movement up or down occurs. When moving up, any values $s \leq p$ are simply discarded. When moving down, all values $s \geq p$ are effectively replaced with a ∞ value, preserving only the count.

Observation 8.1. *Let i be the h-index of S. Then $\forall b \in S$ with $b < i$, the h-index of $S \setminus \{b\}$ is again i and $\forall a \in S$ with $a \geq i$, the h-index of $S \setminus \{a\} \cup \{\infty\}$ is again i.*

Observation 8.1 means that when recursing upwards anything at or below the pivot can be deleted and when recursing downwards all values at or above the pivot can be replaced with a single *overflow* counter, representing ∞. The full algorithm is shown in Algorithm 8.2.

Algorithm 8.2: DHIndex, a recursive approach.

```
   Input: S, o
 1 if |S| = 0 then return o
 2 p ← Pivot(S)
 3 a ← |{s ∈ S : s > p}| + o
 4 e ← |{s ∈ S : s = p}|
 5 b ← |{s ∈ S : s < p}|
 6 if a > p then
 7 |   return DHIndex({s ∈ S : s > p}, o)
 8 else if a + e < p then
 9 |   return DHIndex({s ∈ S : s < p}, a + e)
10 else return p
```

Correctness Observation 8.1 ensures that the filtering in lines 7 and 9 do not modify the resulting h-index on recursion. Lemma 8.1 ensures that the recursion is correct. It then remains to show that any termination of the algorithm will be correct for the given input. If the algorithm then terminates on line 10, then Lemma 8.1 ensures correctness. Otherwise, the algorithm can terminate on line 1. Then, S contains exactly o infinity values, which has an h-index of o.

8.3.2 Choosing a Pivot

The choice of a pivot is crucial for Algorithm 8.2. If the pivot only removes a single element per recursion, the algorithm will run in $O(n^2)$ time. The pivot computation itself has to be fast—linear time or better. A natural pivot is to use the median. This can be computed in $O(n)$ time [30] and results in DHIndex running in linear time.

Lemma 8.2. *Algorithm 8.2 runs in $O(n)$ using the median as a pivot.*

Proof. We need to show that the amount of work reduced at each step is a constant ratio of the input size.

While either recursing down or up, the median value itself is removed along with either all of the values above or below. By definition of the median, this means $|S|/2$ values will be removed. There is $O(n)$ work per recursion, namely first by determining the median and

second by counting high, low, or equal. Hence, we have the recurrence $T(n) = 1 \cdot T(n/2) + O(n)$, resulting in an overall runtime of $O(n)$ [25]. □

The disadvantage of choosing a median is that the computation itself is involved and, while linear time, takes multiple parallel independent passes over the data. Similar to fast selection approaches [210], we implement a simpler pivot selection by simply picking an element uniformly at random.

Lemma 8.3. *Algorithm 8.2 runs in $O(n)$ expected time using a random element as a pivot.*

Proof. Let a *good* pivot be one that results in a constant fraction of the input being filtered out. Then, by [25] we achieve a running time of $O(n)$. Consider the middle $3/4$ of the input values, above the $1/8$-th smallest and below the $7/8$-largest. Any value in the range is a good pivot. Since the length of the range is $3/4$, then a randomly sampled value will be a good pivot with a probability $3/4$. As such, the expected runtime is $O(n)$. □

8.3.3 Parallelizing DHIndex

Given a parallel pivot selection, it is possible to parallelize DHIndex. Assume that S is evenly distributed among the ranks. First, a reduction computes $|S|$, to determine whether to return o early. Next, a parallel fast pivot computation determines p. We use a random pivot and boost the probability by taking multiple samples per processor and finding the median of those samples. Each rank computes its a, e, b values locally, and then a global reduction among those three values is carried out. Each rank then recurses.

Lemma 8.4. *With p processors and assuming an initial $O(n/p)$ distribution, DHIndex runs in $O(n/p)$ computation time and $O(\log n \log p)$ communication time.*

Proof. Each processor only iterates over its data following the pivot. By Lemma 8.2–8.3, the runtime is then $O(n/p)$. The collective operations add $O(1)$ for computation time. For the communication time, there are collective communications performed each round

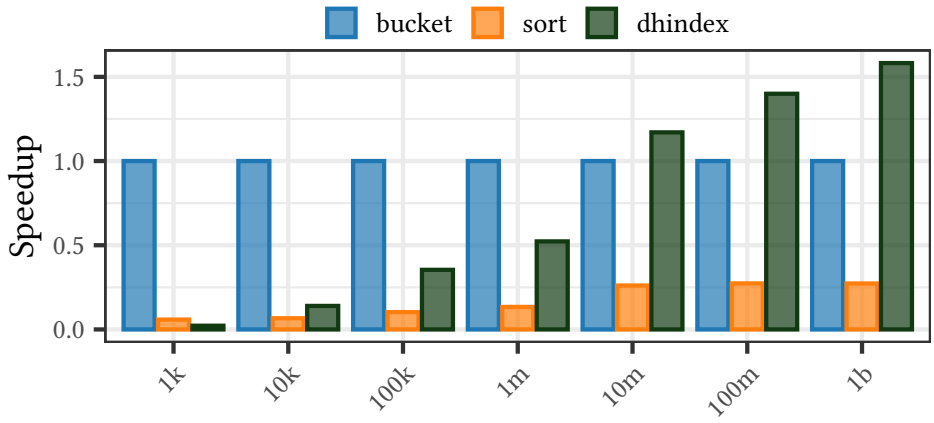


Figure 8.2: Evaluating sequential DHIndex, sort, and bucket.

costing $\log p$ each. In each round, a constant fraction of the input is removed. As such, there are at most $O(\log n)$ rounds, resulting in $O(\log n \log p)$. □

8.4 Implementation and Evaluation

We implement in C++ and evaluate with two systems. For a single node, we use dual-socket Intel Xeon Gold 6140 processors. For distributed scaling, we use a cluster with dual-socket Intel Xeon E5-2683 v4 processors. We synthetically generate data by choosing list elements independently and uniformly at random ranging over the size of the list. We use 5 random inputs and 10 trials for each.

First, we evaluate on a single shared-memory system running sequentially, shown in Figure 8.2. We vary the input size and normalize all algorithms against the fastest prior algorithm, bucket. As the size increases, DHIndex becomes faster than the bucket algorithm.

Next, we study the strong scaling of DHIndex. In Figure 8.3 we scale DHIndex on a single compute node. The slope of the fit line is -0.85 with $R^2 = 0.9859$, close to the ideal slope -1.

We then study the weak scaling of DHIndex across the distributed cluster, keeping the data constant per rank. Figure 8.4 shows the result. Even at 2048 cores, as the problem

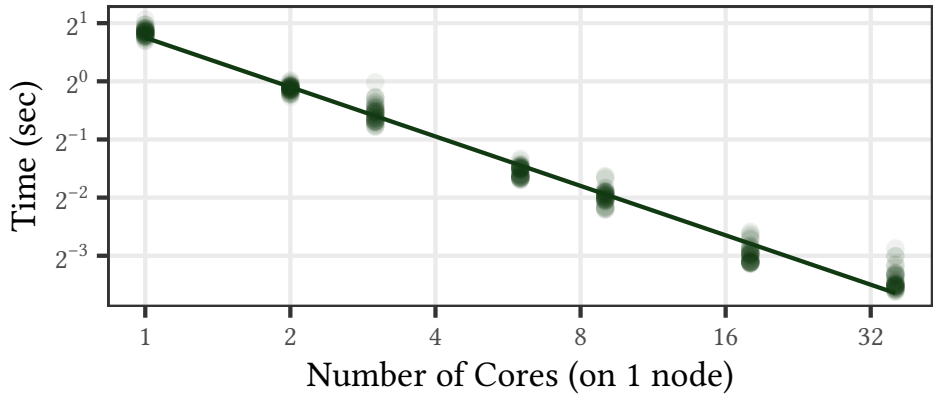


Figure 8.3: Increasing the number of threads available to DHIndex on a single node computing 100 M entries. Even sequentially DHIndex is faster than bucket.

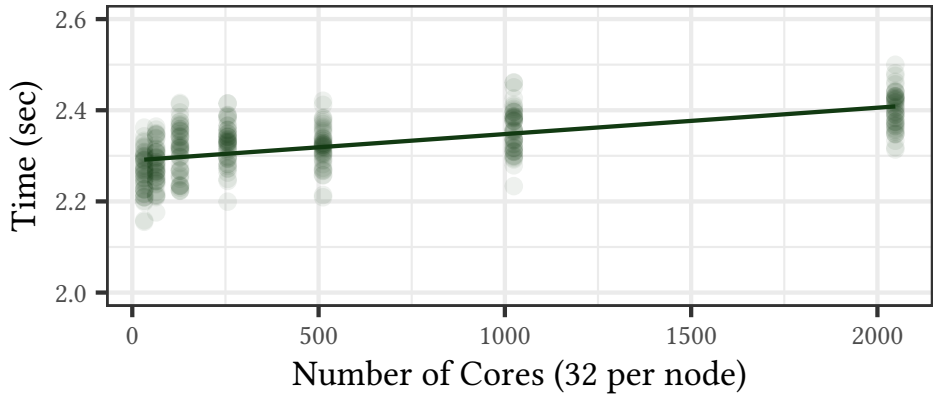


Figure 8.4: Increasing the number of cores while increasing the input size, with 100 million entries per core.

size scales from 3 B to 200 B, the runtime ranges by 0.2 seconds. The slope of the fit line is 0.000058 with $R^2 = 0.4025$, close to the ideal slope 0.

8.5 Summary

We present a new recursive method for computing the h-index that is amenable to parallelization. We implemented our algorithm and experimentally show that it is able to compute h-indices on lists with hundreds of billions of elements in tens of seconds and is faster than sequential algorithms for large inputs. Not only does this algorithm provide performance improvements from existing algorithms, but it enables computation of cores and nuclei in a scalable manner on distributed systems.

CHAPTER 9

SCALING OUT: ELASTIC AND DISTRIBUTED COMPUTATION

Graph analysis is a crucial part of many data analysis pipelines. Numerous graph algorithms and systems have been developed to tackle a large range of problems. Most graph algorithms and systems have been developed as *static* graph systems—they start by loading an input graph, perform some computation, save results, and finally shutdown. Many large graphs are generated from continuous processes, such as website visits, computer network traffic, and more. As graphs have grown in scale, increasing attention has been made towards treating them as *dynamic* graphs, where they change over time and remain in-memory on a system [26]. High performance systems that handle massive dynamic graphs, those in the hundreds of billions of edges, are increasing in importance [40].

There have been dozens of large, distributed graph systems developed over the last decade [184]. In many cases, dynamic graphs are *partially supported*: the underlying system may support some form of updates, and many graph algorithms can be continued from a prior state. Unfortunately, such systems tend to not work well in practice on rapidly changing graphs due to architectural decisions resulting in large start-up and tear-down costs. Many dynamic graph algorithms, especially those which are locally persistent [211, 8] and well-suited for vertex-centric distributed systems [179], may have significant runtime variance between batches of edge changes [77]. Similarly, the rate of change of graphs can vary significantly [209]. A crucial yet overlooked factor in dynamic graph systems is *elasticity*: the ability of a system to rapidly support scaling computational resources up and down, to match demand, meet constraints, or optimize an objective [189, 15, 100, 61, 114].

Why is elasticity important, and especially so for dynamic graph systems? First, dynamic graphs change over time. As they grow, their scale changes, and over time the underlying infrastructure requirements, such as memory, change. Second, graphs can ex-

perience periods of relative calm and periods of significant bursts of changes. Scaling up during bursts and down during calm periods can provide both cost savings and performance gains. Third, even among identically sized batches, some batches necessarily take much longer to process than others [77, 211, 8]. In many cases the cost of the batch is not quickly predicable and so the ability to elastically scale based on the runtime behavior of the batch is important. Fourth, for both static and dynamic graphs, many iterative algorithms require less work for later iterations. Scaling down accordingly can bring significant cost savings [250, 205, 113].

Overall, elastic systems can make trade-offs between runtime performance, memory demand, and cost. Inelastic systems need to continuously consume resources demanded for the peak scale. Elasticity has recently been recognized as an open and important research direction in distributed graph systems for low-latency queries on dynamic graphs [113].

9.0.1 Design Goals

Motivated to provide an elastic dynamic graph processing system for modern, large graphs, we introduce five specific design goals for such a system. Let P be the number of processors in the system.

Goal 9.1. *The system can operate on graphs with hundreds of billions of edges and skewed degree distributions.*

Goal 9.2. *At any point in time, all system participants operate with $O((n+m)/P+P)$ memory, where n and m are the current number of vertices and edges, respectively.*

Goal 9.3. *The system is scalable, and so the only dependence on P for frequent operations is $O(\log P)$.*

Goal 9.4. *The graph is dynamic and can be continuously updated. Maintenance algorithms should optimize for low-latency and low-variance runtimes and should support concurrent queries.*

Goal 9.5. *The system can scale up or down, manually or automatically, during computation to meet demand.*

9.0.2 Contribution

We propose ElGA, a distributed graph system that is able to perform computation on rapidly changing graphs and can do so while the underlying infrastructure scales up and down. Inspired by key-value stores that scale from single nodes to global systems [263], ElGA uses a shared-nothing architecture [242] with a reliable and scalable message passing system. The main question is how to identify which graph edge belongs to which processing agent given a continuously changing graph. ElGA uses consistent hashing [137] to solve this. Prior dynamic graph partitioning approaches that address skewed degree distributions have required information about all vertices in memory [1] (taking $O(n)$ space), which does not meet Goal 9.2 and introduces challenges for processing concurrent graph changes from independent sources. We remove this dependence through the use of sketches [102] where a small, fixed amount of memory contains partitioning information for the whole graph.

Through our design, ElGA meets the above goals. At the same time, our architecture provides better runtime performance than existing static distributed graph systems. We show that distributed graph systems which are both *dynamic* and *elastic* do not have to sacrifice performance or scalability against their static graph system counterparts. We release ElGA as an open source system to encourage development and further improvements to distributed, dynamic, and elastic graph systems[1].

The remainder of this chapter is as follows. In § 9.1 we provide necessary background and a review of existing distributed systems. In § 9.2 we describe ElGA, including its programming model and architectural details. In § 9.3 we perform an experimental evaluation and finally in § 9.6 we summarize the chapter.

[1] ElGA is available at https://github.com/GT-TDAlab/ElGA.

9.1 Background and Related Work

9.1.1 Vertex-Centric Models

Many dynamic algorithms are vertex-centric and communicate over edges in iterations. These are known as *locally persistent* [211, 8]. Within vertex-centric algorithms there are *synchronous* algorithms, where each vertex can only be processed once before receiving updates from all neighbors, *asynchronous* algorithms, where vertices are processed when all necessary neighbor messages are provided, and *partially synchronous* where asynchrony occurs with larger, global synchronous levels [84]. In many distributed graph systems a vertex may be duplicated for load balancing across multiple independent nodes, which we call replicas.

9.1.2 Distributed Graph Systems

Classification of Distributed Graph Systems

There are numerous distributed graph systems serving a large variety of purposes. There are two main dimensions that determine what graph system is suitable for a given application: the properties of the input graph and the desired algorithms. In this section we provide a classification to explain where *dynamic graph systems* fit in.

Starting from the graph properties, the first classification point is whether the input can fit into memory across the system or not. If the graph is too large to fit, streaming graph algorithms [186], sparisification [72], sampling [157], approximations [128], and other randomization strategies, e.g., [5], can be used. We address graphs that can fit into memory across the entire system. The next point is whether the graph is static, i.e., it will not change, or dynamic. If it is static, there are block-based methods, vertex-centric methods, among others [184]. We focus on dynamic graphs. Next, we consider three different algorithm needs. First, some algorithms only perform neighbor traversals or similar subgraph lookups, typically following vertex label constraints. Graph databases are used in

this case [27]. Second, some algorithms require a temporal history of the graph, and so temporal graph systems are used [145, 118]. Third, algorithms may be used for real-time or interactive applications and need results on the most recent graph [26]. Our focus is on this need. In the next section, we drill down based on what the desired latencies of the algorithms are and what kind of scale the graph requires.

Dynamic Graph Systems

There are numerous distributed graph systems that have been proposed and developed and many have some notion of a changing, or dynamic, underlying graph. [26] provides a thorough review of prior systems. Table 9.1 presents a comparative overview of dynamic and elastic graph systems; the columns and systems are described in the following.

Definition 9.1. *A system is* scalable *if it meets Goals 9.1–9.3: it has been shown to run on large graphs, does not require storing all vertices or edges on one node, and does not depend linearly on the number of processors in time-sensitive operations.*

There are many scalable systems and programming models developed for static graphs and with fixed infrastructure, including Pregel [179], Blogel [266], Giraph [110], GraphX [105], and Gelly [41]. These systems provide programming models [184] and, in many cases, robust and flexible code bases that dynamic graph systems develop on top of.

Definition 9.2. *A system is* partially dynamic *if it can re-use prior output to continue an incremental computation.*

Several systems have been developed that are partially dynamic, but do not specifically support low-latency updates. Sprouter [2] and CellIQ [126] address dynamic partitioning costs, but do not address the batch startup costs or elasticity. GraphTau [127] can operate in a partially dynamic manner but was built to provide temporal analysis of graphs. GraM [265] focuses largely on static graphs with fixed partitioning. Delta-BiGJoin [11] is an approach implemented in Timely Dataflow [198] that is optimized for efficient subgraph

Table 9.1: Dynamic and elastic properties of graph systems.

System	Open Source	Scalable	Partially Dynamic	Fully Dynamic	Elastic
Pregel [179]		✓			
Blogel [266]	✓	✓			
Hadoop/Giraph [110]	✓	✓			
Spark/GraphX [105]	✓	✓			
Flink/Gelly [41]	✓	✓			
Spark/GraphX/Sprouter [2]	✓	✓	✓		
Spark/GraphX/CellIQ [126]		✓	✓		
Spark/GraphTau [127]		✓	✓		
GraM [265]		✓	✓		
Timely/Delta-BiGJoin [11]		✓	✓		
Kineograph [46]		✓	✓		
InfoSphere/UNICORN [244]			✓	✓	
Flink/Gelly Streaming [41]	✓		✓	✓	
ZipG [140]			✓	✓	
ChronoGraph [74]			✓	✓	
Naiad [198]			✓	✓	
Kickstarter [256]	✓		✓	✓	
Concerto [154]			✓	✓	
Vaquero et al. [252]			✓	✓	
Spark/GraphX/EdgeScaler [204]			✓		✓
GRAPE [79, 76, 78]	✓		✓		✓
FaRM/A1 [40]		✓	✓		✓
JoyGraph [250]		✓			✓
Graphless [248]		✓			✓
iGiraph [112, 111]					✓
Joker [136]					✓
EIGA	✓	✓	✓	✓	✓

queries. Kineograph [46] operates on snapshots of a dynamic graph and leaves elasticity as a future goal.

Definition 9.3. *A system is* fully dynamic *if it supports low-latency queries and low-latency batch processing, meeting Goal 9.4.*

There are many fully dynamic distributed graph systems, which address the need to run low-latency dynamic graph algorithms. To a large extent these systems have not focused on scalability or elasticity. UNICORN [244] is built on InfoSphere Streams and provides low-latency incremental graph updates, but it has not been shown to scale to large graphs or to provide elasticity. Gelly Streaming [41] focuses on streaming graph algorithms and does not address elasticity or directly support locally persistent algorithms. ZipG [140] provides low-latency queries but focuses on subgraph extraction and does not address elasticity. ChronoGraph [74] is a fully dynamic system built in NodeJS, but not elastic. Naiad [198] is a general purpose dataflow system. KickStarter [256] maintains an approximate computation and re-computes exactly on-demand. Concerto [154] uses elastic key-value stores internally, but does not explicitly support elasticity. [252] re-partitions internally on graph changes but does not address elasticity.

Definition 9.4. *A system is* elastic *if it can rapidly scale its infrastructure, manually or automatically, to meet demand [189, 114].*

There are elastic graph systems, but to date these have not focused on dynamically changing graphs. GRAPE [79, 76, 78] consists of a series of work, some of which address partially dynamic graph algorithms and others which address various aspects of elasticity. The GRAPE model is not scalable as it requires a global view of the graph on a single machine. A1 [40] is a dynamic attributed graph database that is in-memory and communicates over RDMA by building on FaRM [65]. A1 inherits FaRM's elasticity, but is limited in its scope to querying subgraphs and retrieving vertex and edge attributes; it does not support dynamic graph algorithms. JoyGraph [250] demonstrates the advantages of elas-

ticity for static graphs, but does not address dynamic graphs. Graphless [248] operates in the serverless model but does not support dynamic graphs. iGiraph [112, 111] elastically scales during computation to provide cost and performance benefits, but does not support dynamic graphs. Joker [136] is a dataflow system that addresses elasticity, however it has not been developed to address graph problems.

9.1.3 Achieving Elasticity in Clouds

Consistent hashing was developed as a caching technique to decrease hot spots in large, distributed data sets [137]. The idea is to place servers and data uniformly at random in a ring, and then map data to the closest server. Hashes can be used on server IDs and data keys ensure uniform placement of both servers and data. Importantly, when a server joins or leaves, the data mapped to it moves to only a few neighboring servers and all other data has no movement. This effectively provides both smoothness for partition keys and reasonable load-balance across the servers.

Chord [241] was one of the first distributed systems to use consistent hashing. Since then, it has been foundational for many systems that have churn in participants, such as distributed hash tables [134], and in cloud systems that are highly elastic, such as Dynamo, Amazon's key-value store [57]. Consistent hashing has been applied for graphs by treating vertices as keys, providing in high-quality partitioning [1]. However, such approaches only partition vertices or require global information about each vertex.

9.1.4 Sketches

Sketches are tools that allow for approximate results using sublinear space [102]. The first sketch that counts distinct types of elements in a stream is Count Sketch [43]. It works by creating a small matrix and, for each element in the data stream, incrementing or decrementing one cell in each row based on row-specific hashes for the data. The total count is then approximated by the average value of the cells. The CountMinSketch [51]

improves the Count Sketch by only going in *one direction*. Instead of adding or subtracting, CountMinSketch will only add values. A minimum instead of the average is used. For a fixed probability and an additive error of εm after m elements, its space complexity is $\Omega(1/\varepsilon)$. We are not aware of prior uses of sketches in dynamic graph partitioning.

9.2 ElGA

The key to achieving scalability and elasticity in a distributed graph system starts with how the vertices and edges are partitioned among the processing units in the system. Due to the irregular nature of graphs, balancing workloads is crucial. When new compute resources are added or removed, we want to both keep the system well-balanced and to limit data movement, allowing both the graph algorithm runtime and elastic scaling to be quick. Once the partition of each edge is determined, the rest of the system falls naturally into place: any communication through an edge can communicate with the edge owner and vertex-centric programs will work.

Our key contribution consists of a new way of placing the edges given challenging constraints: the graph is dynamically changing, the nodes owning edges are being added and removed, and we cannot use more than a (small) constant amount of global storage.

We solve this by pulling together concepts from cloud computing and streaming algorithms. First, every component in ElGA that needs to be scalable builds on consistent hashing. This forms the backbone of ElGA. Second, ElGA is flexible with receiving messages out-of-order and/or destined for the wrong node. It buffers such messages appropriately and forwards them to the best known destination to achieve eventual consistency. Third, in any situation where global knowledge of the graph is required, we apply sketches. This allows us to globally make decisions about vertices using a small constant amount of memory which can be updated over time.

In the remainder of this section we provide an overview of our system, describe its programming model, and discuss important architectural details.

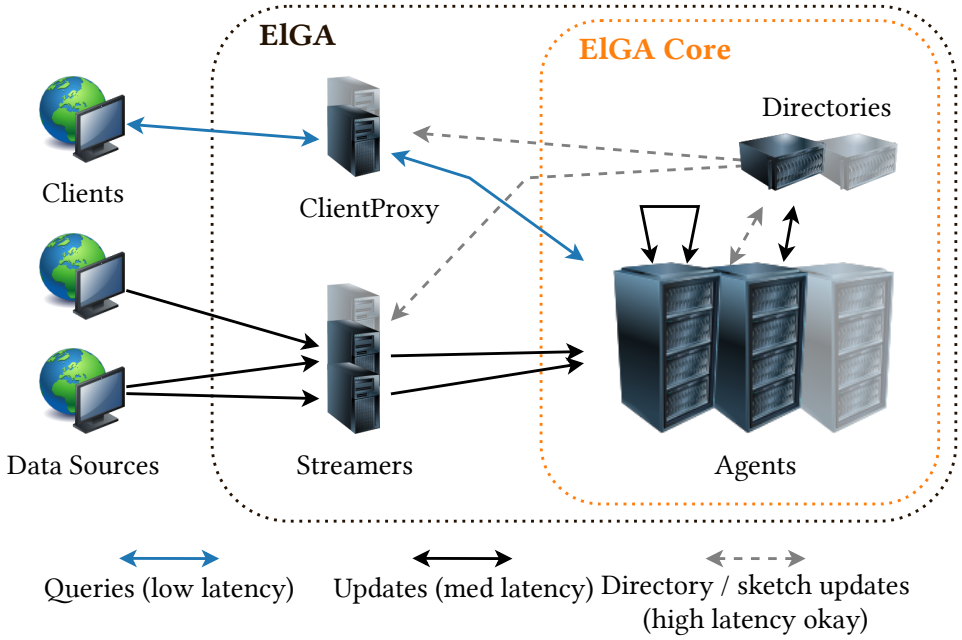


Figure 9.1: An overview of the components of ElGA.

9.2.1 System Overview

A high-level overview of ElGA is shown in Figure 9.1. ElGA follows a shared-nothing design [242]. This means that each entity is single threaded and only communicates via message passing. ElGA is composed of several entities: Agents run graph algorithms and hold graphs in memory; Streamers send graph updates to Agents; and Clients proxy end-user queries to Agents to receive algorithm results. We call Agents, Streamers, and Clients "Participants", as they all need to determine which Agent has ownership of a given edge. The system boundary provides the entry point for other programs to interact with ElGA. We have support for simple TCP and UNIX socket protocols for crossing the system boundary.

Finally, there is a directory system in ElGA which both informs Participants which Agent is responsible for what edge and facilitates synchronization as needed in bulk-synchronous algorithm steps. The directory itself needs to be scalable, as the number of Participants can grow and each Participant establishes a connection to the directory system.

ElGA contains low-latency, medium-latency, and high-latency connections. Low-latency connections need to be returned as quickly as possible. These are used by the clients to query specific algorithm results or properties of vertices. Medium-latency connections consist of updates that are on some critical path for reducing the staleness of a client query. This consists of both updates to the graph and updates required by a batch algorithm to finish computation. Finally, there are connections that allow high-latency which support elastic scaling and load balancing, but are not on any critical path for future client queries.

9.2.2 ElGA Core – Programming Model

ElGA follows a locally persistent dynamic graph algorithm model [211, 8]. Each algorithm is executed when it has some changed state, either a message from a neighbor or replica. The algorithm runs from the perspective of a vertex. It can save local state and send messages along its edges. This is a common model for scalable graph systems, and while there are many alternative models, such as edge-centric [215], subgraph-centric [247], and block-centric [266], this model continues to perform with state-of-the-art results [12].

To support more complicated dynamic graph algorithms than only bulk-synchronous parallel algorithms, such as PageRank, ElGA has support for receiving specific instructions from an algorithm for which edges to expect changes on. If it needs to wait for a result from another vertex, then it places itself in the waiting set for that vertex, at the given iteration. Only when that specific message is received will the vertex be removed from the waiting set. When a vertex is no longer waiting on any messages, it enters an active state and can be processed again.

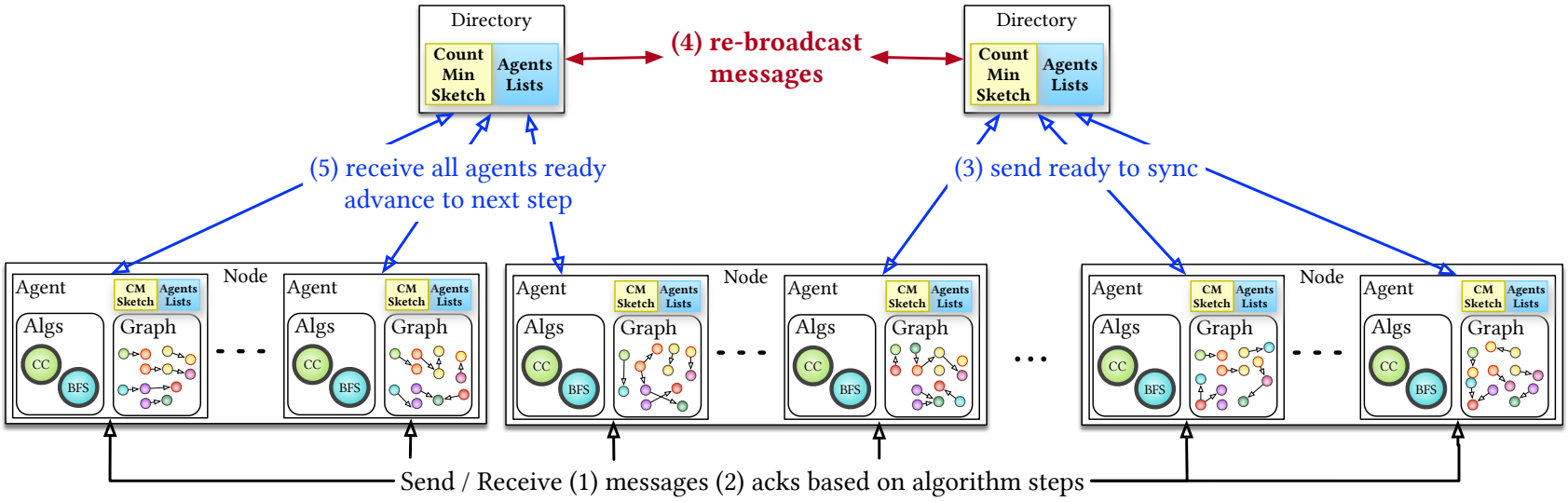


Figure 9.2: Agents sending messages to neighboring vertices and advancing to the next superstep through Directories.

The process of performing an iteration in this model is shown in Figure 9.2. First, **(1)** Agents will send messages—as the algorithm dictates—to various neighbors and **(2)** wait to receive acknowledgements from each. When each internal vertex is inactive **(3)** Agents report they are ready for the next superstep. **(4)** Directories re-broadcast ready messages among themselves. Finally, **(5)** when all Agents are ready, the superstep is advanced and vertices which are marked to become active will begin processing.

9.2.3 Directory System

Inside of the directory system, there are Directories and a single DirectoryMaster (not shown in Figure 9.1). The DirectoryMaster serves as a bootstrap service for ElGA: it is queried once by any component to find a Directory that is open to receiving communication and only queried again if the current Directory leaves the system. When Agents join or leave, or the graph changes enough to impact load balancing, Agents inform their respective Directory server. To keep each Directory in sync, all Directories internally broadcast messages appropriately. This update then propagates throughout the system and eventually, all Participants will use the latest directory. During the gap when some Participants have stale information, messages will be forwarded.

The full broadcast size is $O(P + dw)$, as each Agent's IP address and connection information are sent, P is the number of Agents (one Agent per core), d is the CountMinSketch depth, and w the CountMinSketch width. Each Directory also keeps track of the current batch ID, a monotonically increasing clock used to bootstrap Agents and ensure consistency. As shown in Figure 9.2, Directories are also used to facilitate global synchronization by waiting for their connected Agents to be ready and broadcasting state changes.

Sketch Size

As sketches are broadcast throughout ElGA, we want to ensure that they are appropriately sized. For a given number of edges, we can compute the size desired for a given error at a probability. Consider a point in the stream with m edges seen. Let d_i be the degree of vertex i at this point and $\hat{d}_i$ be the estimated degree. There are two parameters: the width of the table and the depth. The width can be set to $\lceil e/\varepsilon \rceil$ and the depth $\lceil \log 1/\delta \rceil$, and the sketch guarantees an additive error $\hat{d}_i \leq d_i + \varepsilon m$ with probability $1 - \delta$. If a graph will have 100 billion edges, we can get a 99.965% probability of each degree estimation within just over 1 million if we use a width of 2^{18} and a depth of 8. If the vertex replication threshold is 2 million, this table size will suffice and fits in 8 MB. We explore various table sizes in § 9.3.5.

9.2.4 Agents

Agents are responsible for holding the graph in memory and carrying out the computation on the graph. They provide the vertex-centric model for algorithms to run in and manage all operations on components of the graph.

Agents themselves are relatively straightforward. They operate as a state machine and, during computation, either execute the algorithms on their vertices, send updates to other Agents, or receive updates from Agents. They continuously poll on their communication channel and act on whatever packet they receive. First, the packet is inspected to ensure that the endpoint remains valid. If the receiving Agent is no longer the correct destination, the packet is forwarded to the latest, correct Agent. Each packet with edge data may contain iteration information. If it is for an iteration in the future, the packet is stored until the computation can catch up. In ElGA's asynchronous mode, vertices are individually processed when they no longer have any outstanding updates they are waiting on. In the synchronous mode, each vertex must have no outstanding updates. In between supersteps, some vertices may have updates that are sent to their replicas, in the case that the vertex

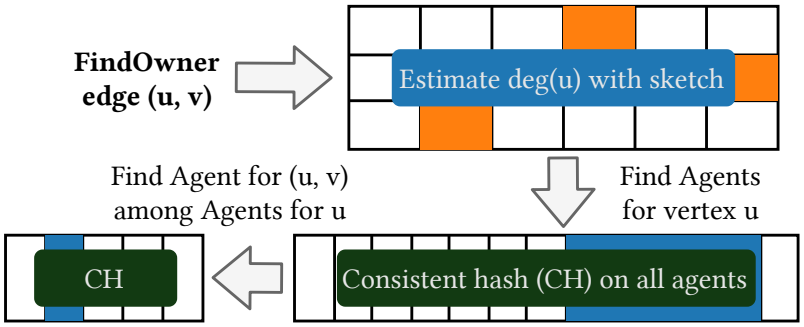


Figure 9.3: Participants look up Agents for edges by estimating the degree and then applying two consistent hashes.

is high-degree and split across multiple Agents. Garbage collection of received neighbor values and other book-keeping data occur at superstep boundaries.

While a batch is running, the graph does not change: any edge changes are buffered. Once the batch is over, these updates can be processed and the Agent becomes ready to perform a computation again. Depending on the algorithm, only updated (or *active*) vertices will be processed at the next iteration.

Finding an Edge

Each Participant receives enough information from its Directory to find the owner of any edge. This process is the heart of ElGA's load balancing scheme, and it enables dynamic graphs and elastic scaling. This process is outlined in Figure 9.3.

A participant will first perform a query into the CountMinSketch to identify how many Agents are responsible for handling the given vertex's edges. This is a biased approximation as it may exceed the degree but not underestimate it. A participant will then use a consistent hash to find the starting Agent and return Agents for the number of replications, based on the degree. It will then perform a second consistent hash on those Agents to find the final Agent that is responsible for the edge. Each consistent hash can be implemented with a binary search over the Agent list taking $O(\log P)$. Querying the degree estimate takes $O(d)$, where d is typically 8.

Concretely, each consistent hash consists of the following. Each Participant has a list

of all the Agents. Each Agent ID is then hashed and added to a vector. To query for a vertex, the vertex ID is then hashed and the next highest Agent in the vector is selected as its owner. If the CountMinSketch indicates a replication factor of $k > 1$, then the Participant needs to select between the next k-highest Agents in the vector. In this case, a second hash is performed on the destination of the vertex to choose which of those k Agents.

For efficiency reasons, if only *some* Agent responsible for the vertex is required, e.g., for a vertex query, then the last consistent hash is bypassed and one replica is chosen at random.

Virtual Agents

Despite using two-levels of consistent hashing and sketching, creating only as many Agents as hardware cores still yields some load imbalance. To address this problem, we introduce *virtual agents*. Instead of putting a single ID into the vector, an Agent will put a variable number of IDs into the vector. We experimentally determined 100 as a good number (see § 9.3.5). This significantly improves the load balance but increases the lookup time by a constant factor (O(log 100)). Future work could explore dynamically adjusting the number of virtual agents over time based on memory or computation pressure or for heterogeneous systems.

Agent Elasticity and Sketch Updates

If an Agent joins or leaves, or receives an update to its CountMinSketch, it needs to ensure that it does not have any edges that need to migrate to another Agent. In ElGA, we take a straightforward approach of re-computing the correct destination for all current edges. Any edges that are in the wrong Agent are removed and forwarded appropriately. No state change can occur until receiving acknowledgements.

When an Agent leaves, it only signals it is leaving to the Directory. After receiving

the next directory update, it evaluates its edges normally and determines that they all need to leave. Only when it has no edges and has waited a period of time will it disconnect.

To handle fully elastic autoscaling, ElGA comes with an API for metric collection and autoscalers. We implemented a simple reactive autoscaler that computes the exponential moving average of a metric and scales to the average divided by a scaling factor. We implemented Agent metrics for graph change rates, client query rates, and superstep times. Metrics are passed to Directories.

9.2.5 Communication

Communication in ElGA occurs using ZeroMQ [115]. ZeroMQ is designed as a networking layer that promotes various communication patterns. For example, it provides a publish/subscribe pattern, where multiple subscribers can receive select updates from a publisher. Internally, ZeroMQ handles message routing and resiliency. We configure ZeroMQ to use TCP between nodes and its interprocess protocol within a node. ZeroMQ operates on separate threads, allowing for overlapped computation and communication management, including packet routing, buffering, and filtering.

ElGA uses a simple serialization and deserialization protocol on top of ZeroMQ messages. The first byte of any message is a packet type which determines how a Participant will handle the message. ElGA's protocols typically involve direct memory copies into ZeroMQ's network buffers.

For low-latency messages, we use the `REQ/REP` model, which is designed for blocking requests and responses.

For messages with medium-latency, we use the `PUSH` communication pattern, which is a non-blocking send. This allows the client to continue executing while ZeroMQ finishes sending the message. This can be crucial when there is a large backlog of messages or potential networking issues. In cases where an explicit acknowledgment is required, a second `PUSH` is then sent in return.

Finally, we use the `PUB/SUB` model for high-latency messages. Here, Participants will subscribe to the messages they are interested in: Agents subscribe to synchronization barriers from Directories, all Participants subscribe to directory updates, and so on. As ElGA uses only a single byte for the message type, filtering subscriptions in ZeroMQ is efficient. ZeroMQ internally will route, filter, and duplicate the messages as intended.

There is an overhead with ZeroMQ: using Mellanox ConnectX/5 NICs, we benchmarked the latency of an MPI send at around 1 μs, a raw TCP send at 4 μs and a send through ZeroMQ at over 20 μs.

9.3 Experiments

In this section we describe our experiments to understand and evaluate the design decisions and performance of ElGA. Following standard distributed graph system experimental methodologies [75], we run five independent trials for each experiment. We report the means and, assuming a t-distribution as the sample size is small, we show the 95% confidence intervals for the mean. All results were checked for correctness among the baselines and ElGA, and, when applicable, against ground truth.

We implemented ElGA in C++17 and use ZeroMQ 4.3.4. Our dynamic graph is stored as a flat hash map with vectors. We configured all systems to use 64-bit integers for vertex IDs. We compiled with GCC 8.3.1 using `O3`. We store both in and out edges.

9.3.1 Experimental Environment

Our experimental environment consists of 65 servers, each with dual-socket Intel Xeon E5-2683 v4 processors with 16 cores each, 512 GB RAM, and three local SSDs. One server is dedicated to filesystem and other metadata, and so 64 servers are used for computation. Each has Mellanox ConnectX 5 NICs and are connected to an Arista 7500E switch, which supports 100 Gbps. Jumbo frames were enabled. We run Ceph [261] on each node to provide a distributed filesystem for storing edge lists and auxiliary information and we

run HDFS [36] for the state-of-the-art graph systems that we used as baselines, as they depend on it. None of our experiments compare graph loading times and and so the use of HDFS or Ceph is not being evaluated; we chose Ceph as it achieves higher performance on our system, however the baselines are implemented with HDFS. We use CentOS 8 and performed basic system tuning using `tuned`'s `hpc-compute` profile.

9.3.2 State-of-the-art Baselines

While there are no known dynamic and elastic baselines, we evaluate against Blogel [266] and GraphX [105]. We chose Blogel as it has been shown to be the state-of-the-art static system [12] and we chose GraphX both as it exhibits strong performance and is the platform for many snapshot-based partially dynamic systems (see § 9.1.2). We confirm previous results [12] that show that the Voronoi partitioning variant of Blogel (which we call Blogel-Vor) is not competitive against their simple vertex partitioning implementation (which we call Blogel) or even GraphX, even when partitioning time is ignored. Unfortunately, Voronoi is the only partitioning strategy implemented for the algorithms in our experiments.

We extensively explored the performance of both Blogel and GraphX in terms of MPI libraries and Java systems, Java garbage collection parameters, and more. We found that using dedicated SSDs for scratch storage for GraphX, along with the G1 garbage collection operating in parallel, a dynamic number of executors, and initial heap allocations of almost all available memory allowed GraphX to be competitive against Blogel and other systems. We configured GraphX to use its three main built-in partitioning strategies. We found that Java 8 provided the fastest runtimes for GraphX, although the loading times were faster with Java 11. As such, we use OpenJDK Java 8 throughout. For Blogel, we tried different compilers and MPI libraries and conclude that using OpenMPI 3.1.4 and GCC, as suggested in the original papers, still provides the fastest end-to-end runtimes. We use GCC 8.3.1 with `O3`. We found using 8 MPI ranks per node for Blogel was fastest, whereas GraphX was

fastest with per-stage dynamic executors choosing the number of cores from 1 to 64 during runtime. We used the best found settings for all shown experiments.

Together, these two baselines represent the *hardest cases* for ElGA—if we compete well against them, we show that our design and implementation which supports dynamic graphs could even be used for end-to-end improvements in static cases. Our experiments show this is indeed the case. We stress that we do not include loading, partitioning, or saving time, as the baselines are not built to handle dynamic graphs and have unoptimized code in each area.

9.3.3 Static and Dynamic Algorithms

We evaluate with two iterative vertex-centric algorithms commonly used in distributed graph system benchmarks [12] that are implemented on Blogel, GraphX, and ElGA: PageRank and weakly connected components (WCC). In PageRank, at each iteration, a vertex receives messages from each in-neighbor, aggregates them with a sum, scales the value, and sends its values out to its out-neighbors. In WCC, a vertex aggregates and sends with a minimum instead of a sum and only sends updated minimums, but to both in- and out-neighbors. In the static case, WCC initializes each vertex to a unique identifier. In the dynamic case with insertions, known as the incremental case, each vertex retains its old or initial component information and only vertices directly modified in the batch are activated. When a vertex receives a message, it becomes active.

To effectively evaluate the graph systems, we ensured that all algorithms are the same across each system, including termination conditions. As such, the performance differences come from the systems themselves. We ensure our implementation's correctness by comparing against the baselines and ensured floating point values were correct up to 10^{-8}. For both baselines, we used the respective algorithms distributed with the baselines, from the original authors. We observed each system perform the same number of supersteps.

While PageRank and WCC are common graph algorithms, important future work in-

Table 9.2: The graphs used in our experiments.

Graph	A-BTER Scale	n	m	EL Size
Twitter-2010 [152]	-	42 M	1.5 B	25 GB
Friendster [267]	-	65 M	1.8 B	31 GB
UK-2007-05 [32, 33]	-	105 M	3.7 B	63 GB
Datagen-9.3-zf [75]	-	555 M	1.3 B	34 GB
Datagen-9.4-fb [75]	-	29 M	2.6 B	65 GB
Email-EuAll [159]	×5000	1.3 B	5.6 B	105 GB
Skitter [159]	×200	339 M	6.3 B	119 GB
LiveJournal [17, 161]	×100	484 M	8.6 B	161 GB
Amazon0601 [156]	×2000	807 M	9.8 B	183 GB
Graph500-30 [42, 197]	-	448 M	17 B	319 GB
Gowalla [47]	×10000	2.0 B	28 B	568 GB
Patents [159]	×1000	3.7 B	33 B	673 GB
Pokec [245]	×1000	1.6 B	44 B	898 GB
Pokec [245]	×2500	4.0 B	112 B	2.3 TB

cludes studying the performance of other algorithms in ElGA which exhibit different bottlenecks and communication patterns [125].

9.3.4 Datasets

ElGA is designed to address large graphs which cannot fit in shared memory. As such, we focus our evaluation on large datasets. We use datasets from LAW [32] and SNAP [160]. We evaluate on representative datasets from social networks, web crawls, product purchases, location, citation, and email data. We further test with the largest published and available synthetic graphs from the LDBC Graphalytics benchmark [125]: the Graph500 30-scale graph [42, 197] and the largest currently published Datagen `fb` and `zf` graphs [75].

In most cases, the available datasets are not as large as modern commercial or real-world datasets and these publicly available graphs are all easily run on a single, shared-memory system. To evaluate ElGA's performance over the variety of datasets it is intended for, along with the intended scale, we use A-BTER [238]. This takes an existing graph—that can fit into shared memory—computes degree and clustering coefficient distributions,

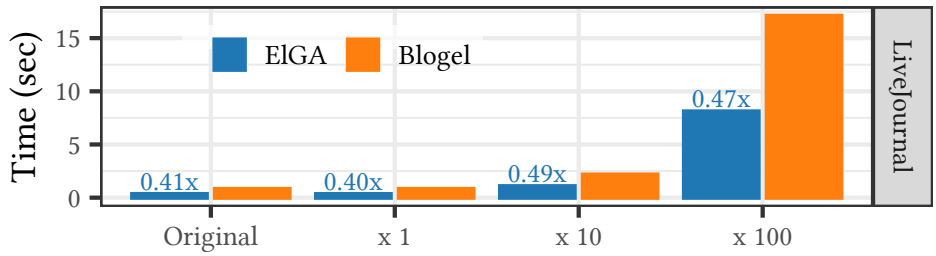


Figure 9.4: Per-iteration runtime of PageRank on LiveJournal with three A-BTER generated LiveJournal-like graphs. The relative runtimes (shown in blue), i.e., ratio between ElGA's and Blogel's runtimes remain consistent.

and then generates random *scaled up* graphs that share the same distributions and properties. It is important to test on larger datasets as unexpected inefficiencies and complications can arise at scale. By using A-BTER, we are able to experiment on a variety of representative graphs *at scale*. Table 9.2 shows the datasets used in our experiments.

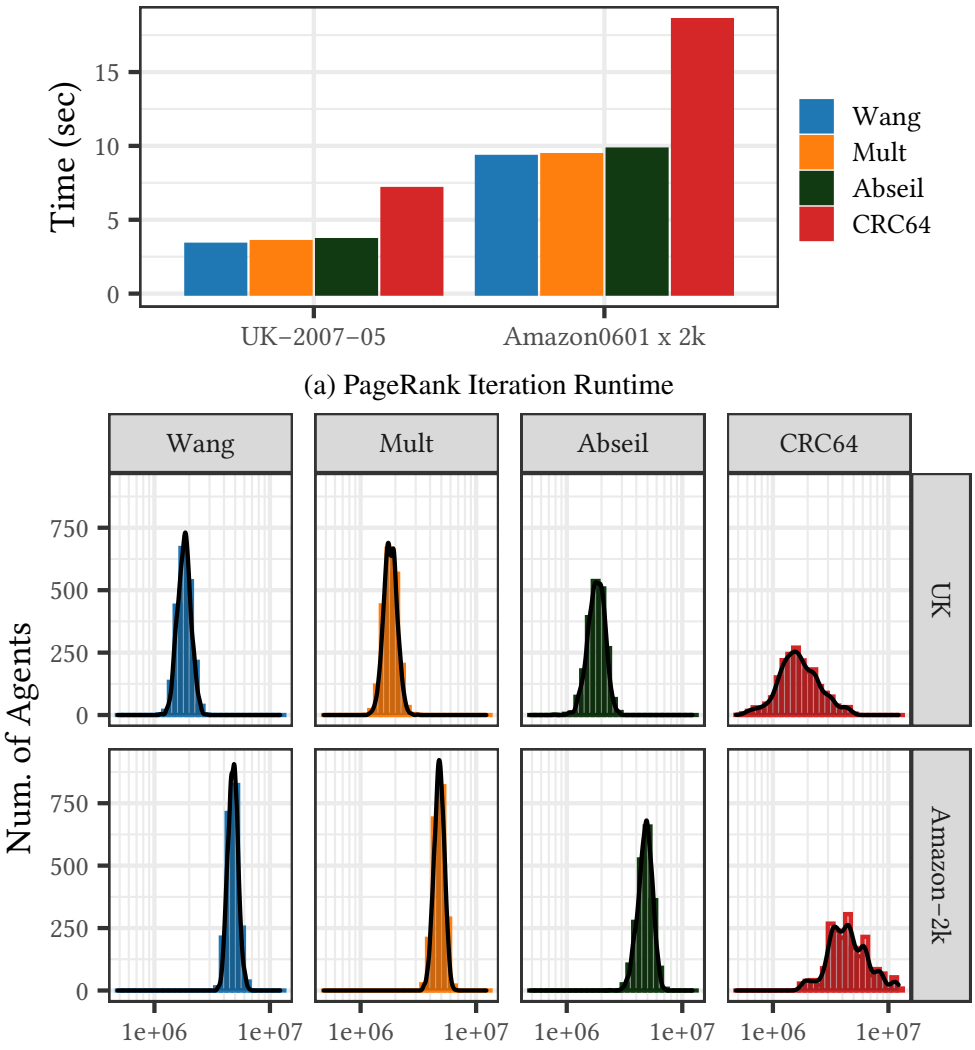
To understand how performance may change with A-BTER synthetic graphs, in Figure 9.4 we show the runtime on the original LiveJournal [17, 161] graph, a synthetic version with no scale difference (×1), and two scaled-up versions (×10 and ×100). While there are some relative differences in system performance as the graphs' scale increases, the runtime at the same scales matches well for each system. These results extend to the other graphs we tested, showing that A-BTER is a valuable tool for evaluating performance of graphs with varying structure at larger scales.

We extended A-BTER to stream edge updates, allowing ElGA to directly receive the graph as it is generated.

While all of these graphs are temporal and dynamic, the given datasets do not have edge deletion or insertion information. As such, we model their dynamic change by first deleting a random sample of edges and second adding the sample back in, as a batch.

9.3.5 Design Choices

We want to understand the impact of various design choices in ElGA. First, we look at the hash function. The hash function plays a crucial role: it is used, along with virtual Agents



(a) PageRank Iteration Runtime

(b) Histograms of Edges per Agent

Figure 9.5: The hash function has a large impact on the runtime. We found that Wang's 64-bit integer hash performs the best. The runtime performance follows the quality of the edge distributions. Ideal is a single vertical line.

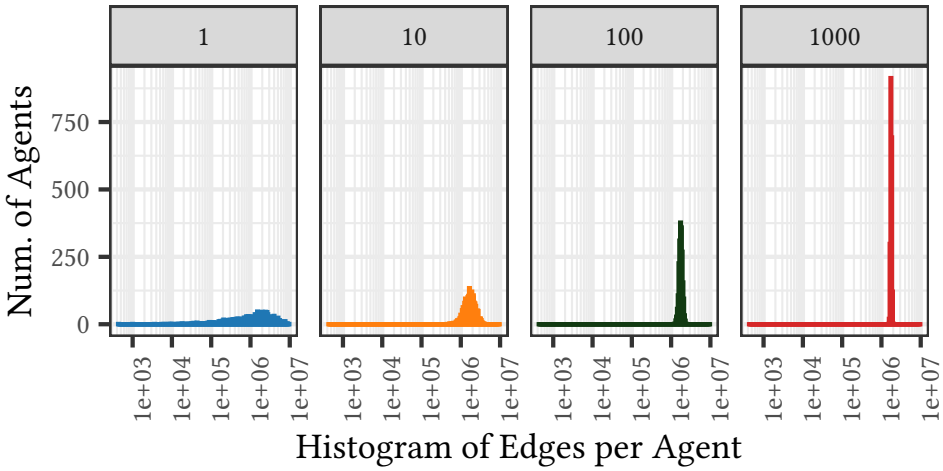
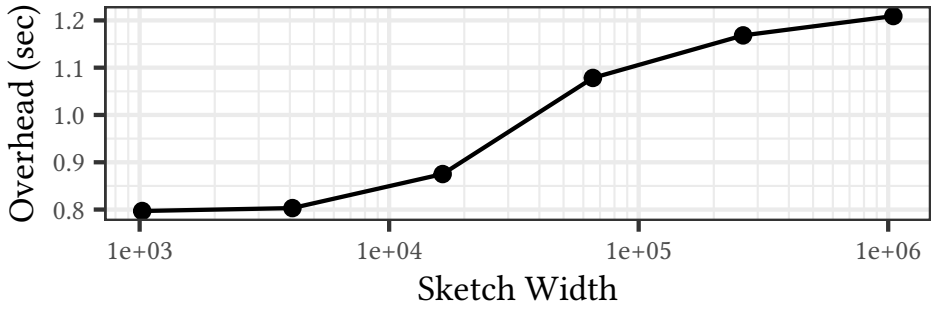


Figure 9.6: The load balance distributions for 2048 Agents as the number of virtual agents per Agent is varied from 1 to 1000 for Twitter-2010. Beyond 100 improvements do not outweigh the computational cost (not shown here).

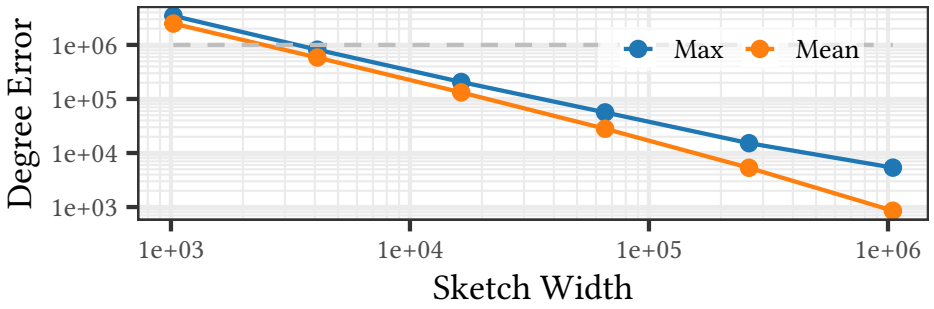
and the CountMinSketch, for every edge *access* to find the corresponding Agent. It needs to be both fast and of high quality, that is, result in a uniform distribution. We evaluate different hash functions in Figure 9.5. Mult is from [240], Abseil is a non-deterministic hash similar to Mult used in the Abseil C++ library, and CRC64 [66]. Due to the number of calls to the consistent hashing system, we use a 64-bit ring and avoid more expensive hash functions, such as the commonly used cryptographic ones [57]. Thomas Wang's [259] is the best performing hash function tested and we use it for the remaining experiments.

Second, we show the impact of the number of virtual agents. With more virtual agents, a larger number of nodes will need to receive or send new edges when an Agent leaves or joins and each consistent hash lookup is more expensive. However, the number of edges per transfer decreases and the load balance improves. In Figure 9.6, we show the load balance distribution for Agents as the number of virtual agents *per Agent* is varied for Twitter-2010. We found a similar behavior for all graphs. At 100 virtual agents per Agent the distribution both has a high quality and sufficiently low lookup rates, and so we use this value for all graphs.

Third, we evaluate our CountMinSketch parameters. Our sketch approach enables high-degree vertices to be split across multiple Agents within ElGA. Each split incurs an



(a) Overhead per PageRank Iteration on Twitter-2010



(b) Maximum and Average Degree Errors on Twitter-2010

Figure 9.7: The runtime cost of resolving edges to Agents along with the degree estimation error as the table width varies. With a replication threshold even as low as 2 million, any max degree error below the dashed line (at 1 million) means the sketch will result in no replication error.

overhead, and so we only want to target vertices that cause significant load imbalance or memory pressure and reduce the number of unnecessary replications. In Figure 9.7, we explore the CountMinSketch width parameter. We vary the width and compute both the runtime overhead per PageRank iteration and the maximum and average error in degrees. We set the threshold high for replicating, at 10^7, and so we can use a small sketch size of $10^{4.2}$, below the inflection point for added overhead and without replication error across our datasets.

9.3.6 Scalability

We next study our scalability and show that the shared-nothing architecture is able to both strongly and weakly scale. We focus on PageRank and show that with ElGA it scales up, both with more cores per machine and more machines. In Figure 9.8, we vary the number

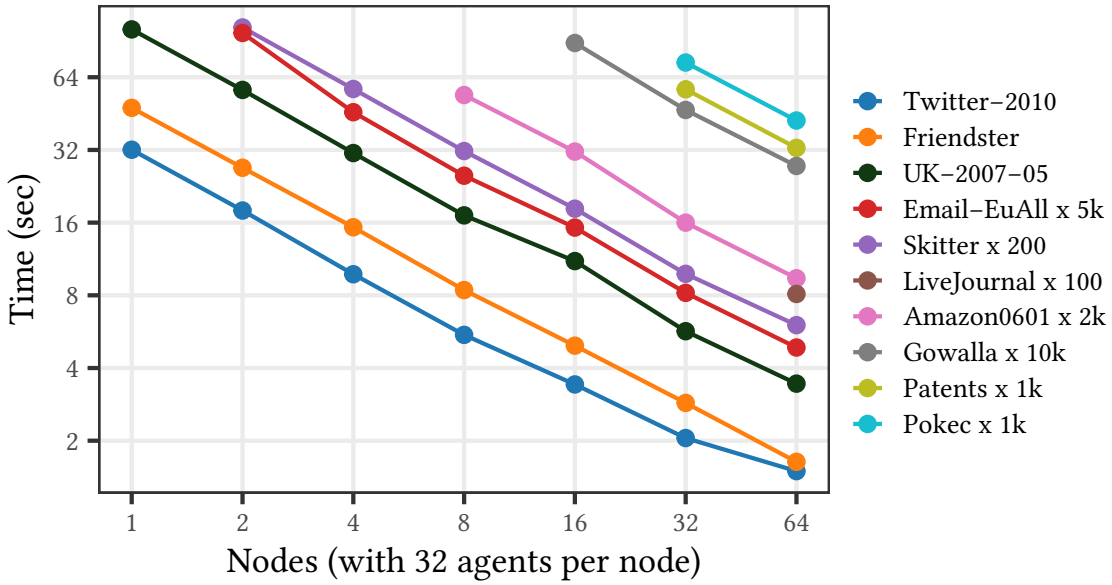


Figure 9.8: The scalability of ElGA reporting PageRank iterations as the number of nodes are varied. The larger graphs run out of memory on small numbers of nodes.

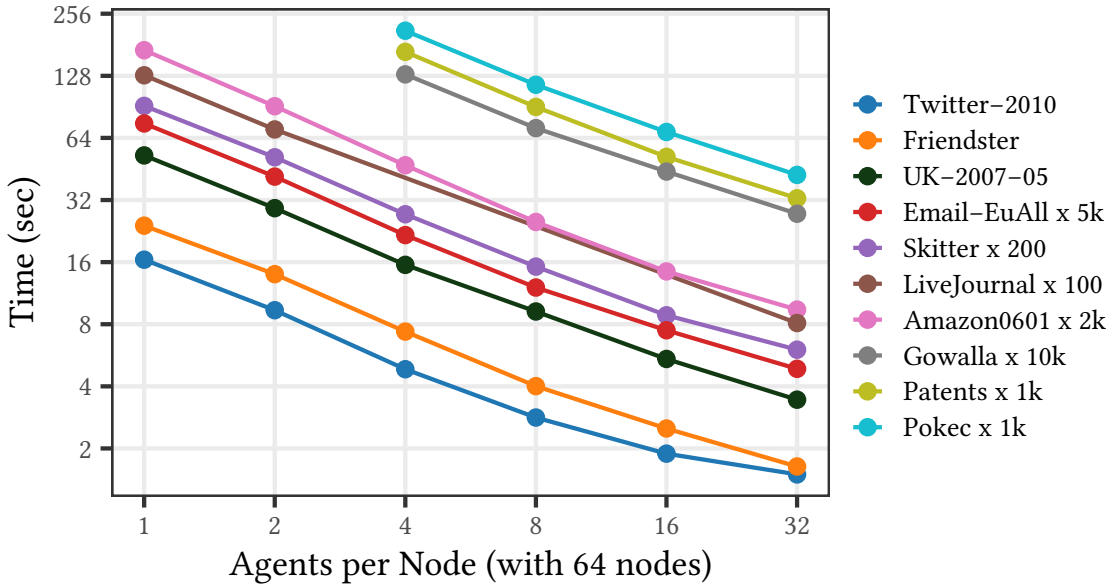


Figure 9.9: The scalability of ElGA reporting PageRank iterations as the number of Agents per node are varied.

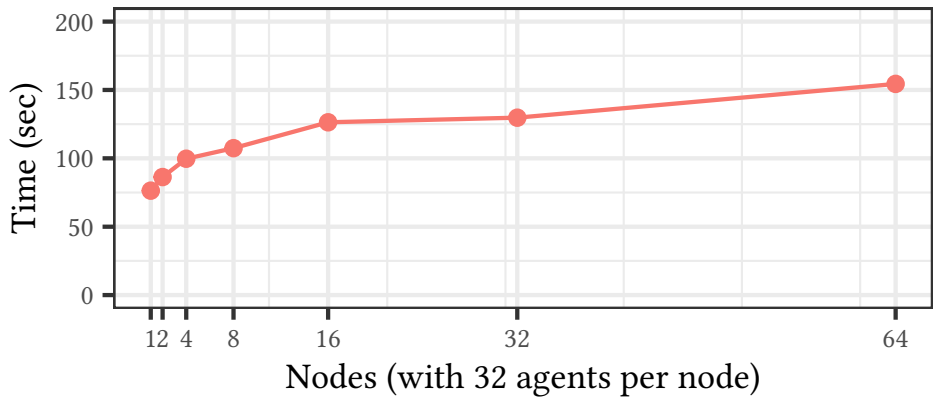


Figure 9.10: ElGA's weak scaling with the Pokec dataset. The scale ranges from ×39 to ×2500. A horizontal line is ideal.

of nodes and report the per-iteration PageRank time. For each graph, adding more nodes results in lower runtimes. As more memory is required per node with fewer nodes, the large graphs stop fitting into memory and so we cannot report their runtimes. In Figure 9.9, we keep the number of nodes fixed at our cluster size, 64, and instead vary the number of Agents that we run on each node. Similarly, adding more Agents results in faster runtimes.

Next, we look at weak scaling. We show the per-iteration runtime of PageRank as we scale the Pokec dataset from 1.7 billion edges to 112 billion, while keeping the degree and clustering coefficient distributions within 2% error. The results are in Figure 9.10. With only a few nodes, and a significantly reduced amount of communication, ElGA performs better per edge than itself at a larger scale. However, at these scales many real-world graphs will not fit into memory. Above 16 nodes our scaling is close to ideal, a horizontal line.

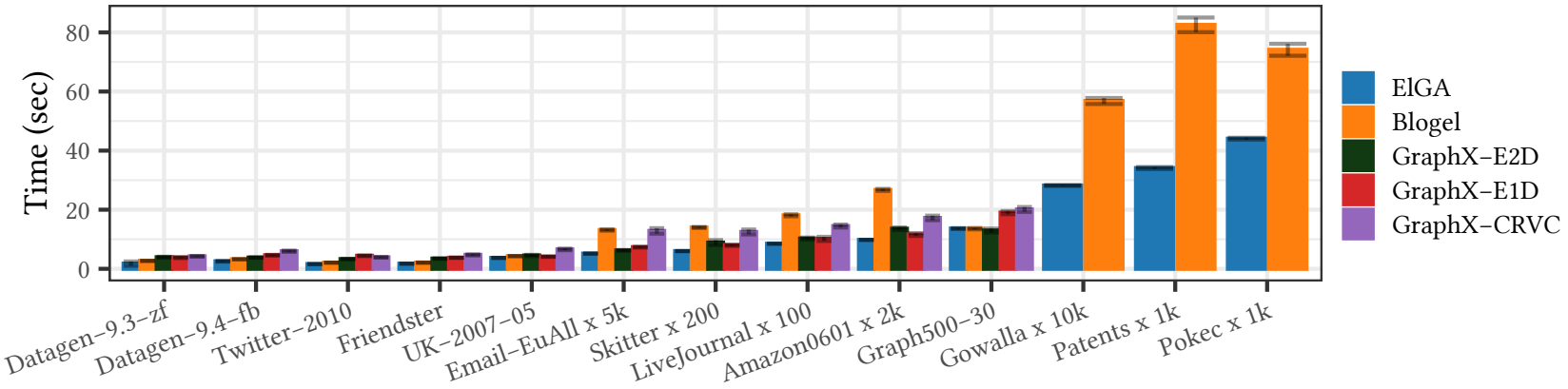


Figure 9.11: ElGA's per-iteration PageRank runtime compared against Blogel and GraphX, using 64 nodes. GraphX includes a significant partitioning overhead (not shown here) and ran out of memory on the larger graphs. A t-test shows ElGA is fastest with $p < 0.0005$ in all datasets except for Graph500-30, which is inconclusive with $p > 0.05$. 95% confidence intervals are shown.

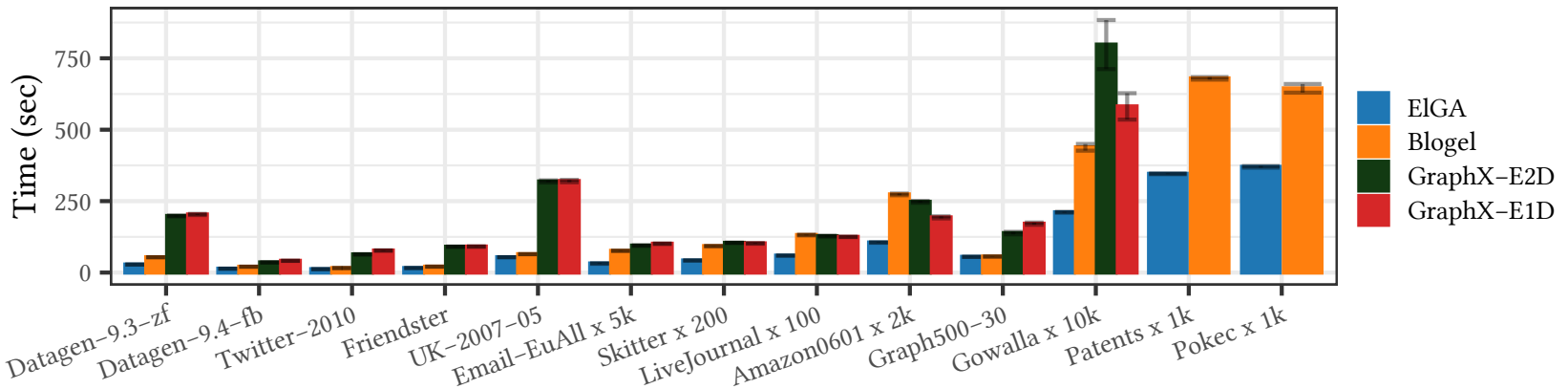


Figure 9.12: The weakly connected components runtime for ElGA, Blogel, and GraphX. In all cases, a t-test shows ElGA is fastest with $p < 0.0005$ (except Graph500-30, where ElGA is fastest with $p < 0.03$). 95% confidence intervals are shown.

9.3.7 Comparison with Static State-of-the-art

We compare PageRank iterations against *static baselines* in Figure 9.11. Given Blogel-Vor's poor performance against Blogel we do not show it in the results. GraphX runs out of memory on the largest graphs. ElGA is designed to handle a constantly and rapidly changing graph, yet we outperform the baselines even when ignoring partitioning time and other static costs of those systems. This is a surprising result. Both Blogel, the second fastest, and ElGA use C++. Blogel uses a CSR internally to hold the graph which is faster than our flat hash maps (but do not easily support dynamic graphs). Further, Blogel uses MPI, and as we showed in § 9.2.5 MPI has 20× lower packet latencies on our cluster. The underlying filesystems's I/O performance is not included in the timing results here.

As our algorithms are the same, we attribute our performance to an interesting property not well explored in distributed and scalable graph systems: even with bulk-synchronous parallel algorithms, allowing messages to arrive out-of-order, with communication handled in separate threads, keeping global state limited, and using a shared-nothing server-based polling system inside each Agent provide significant benefits. We are able to accommodate the increase in computation with per-edge Agent lookups as Blogel is fastest with 8 cores per node, likely due to MPI allreduces saturating the network, and we take advantage of all cores.

To evaluate weakly connected components, we had to fix a bug in Blogel where it does not consider an undirected form of the graph. We did this by symmetrizing the input graph (which along with other I/O costs is not shown). The results are shown in Figure 9.12. We were not able to run GraphX with CRVC partitioning as it ran out of memory on almost all graphs. Similarly, Blogel-Vor did not provide competitive results and so is not shown. All of our results show better performance for our tuned baselines than experiments reported in [12] (with Haswell, not Ivy Bridge CPUs and more RAM).

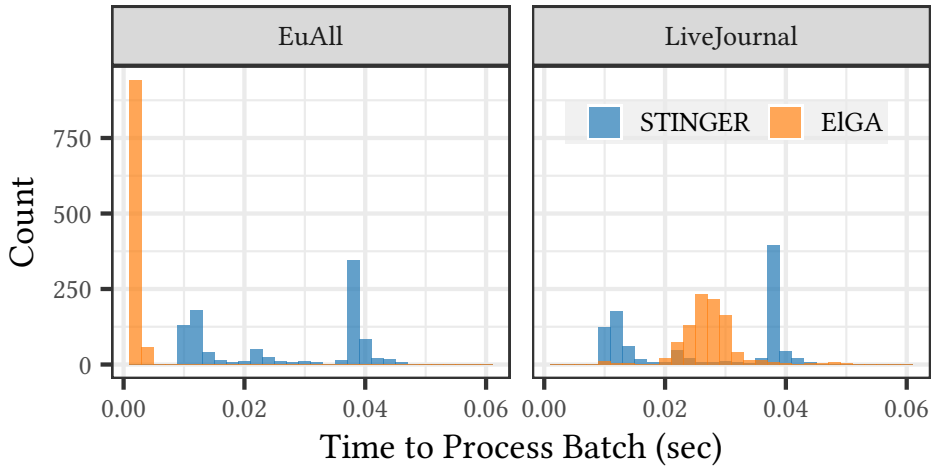


Figure 9.13: ElGA and STINGER maintaining components.

9.3.8 Comparison with Single Node Systems

We next consider the COST [187] of running ElGA on a single node compared with a specialized inherently shared-memory algorithm and system. We are only aware of one publicly available implementation of a dynamic WCC, which is part of STINGER [67]. We were unable to run STINGER on billion-edge graphs; we instead run LiveJournal and EuAll at their original scale. In Figure 9.13, we show the insertion of the last 1000 edges. STINGER can likely optimize for some easy batches due to its global view. It has a bimodal distribution that is surprisingly similar across graphs. For LiveJournal, ElGA's median runtime is 0.027 seconds; STINGER's is 0.032. Note that STINGER does not include any distributed support and uses OpenMP to parallelize. We also compared with GAPbs [23], a shared-memory parallel static graph system. GAPbs takes 0.94 seconds, including building its CSR from an in-memory edge list and running WCC.

9.3.9 Dynamic Behavior and Elasticity

Even though for static cases it scales and performs well, ElGA is primarily designed to be both *dynamic* and *elastic*. An important consideration for a dynamic graph system is the rate of edge changes it can accept. In Figure 9.14 we show the edge insertion rate for

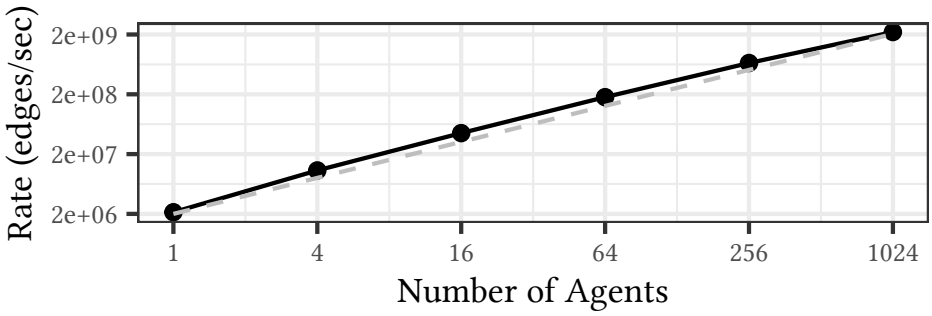
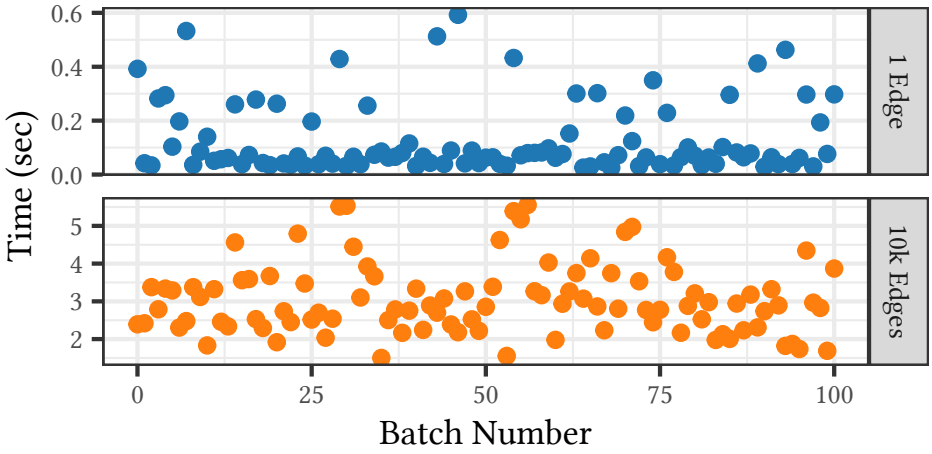
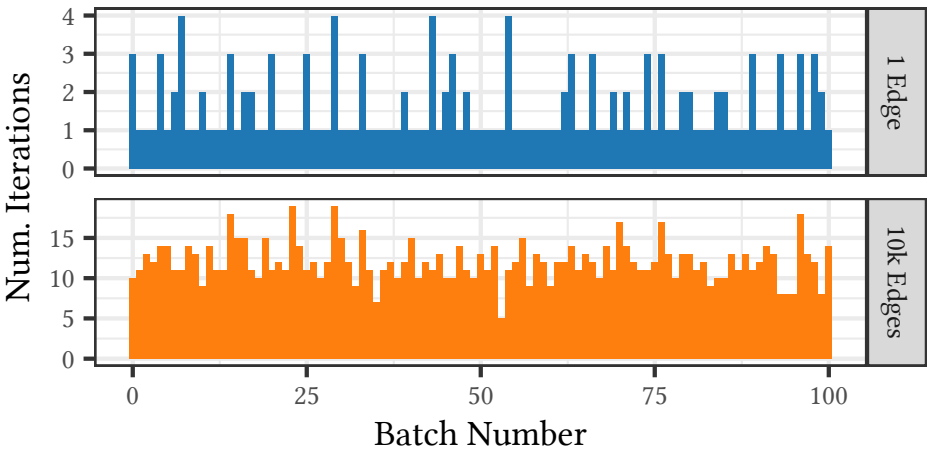


Figure 9.14: The insertion rate of edges from Skitter. Agents densely fill nodes, so 64 Agents run across two nodes. The dashed line shows ideal linear scaling.



(a) The runtime for each batch. Note differing scales.



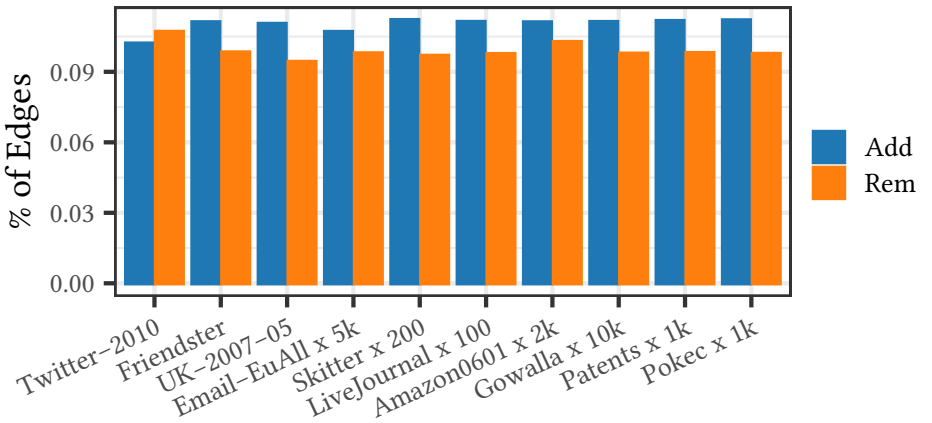
(b) The iterations until convergance for each batch. Note differing scales.

Figure 9.15: Maintaining connectivity on Twitter-2010. From scratch, ElGA takes 14 seconds. (Not shown here: GraphX takes over 49 seconds for one iteration, due to start-up/teardown costs; its recomputation takes 66 seconds.)

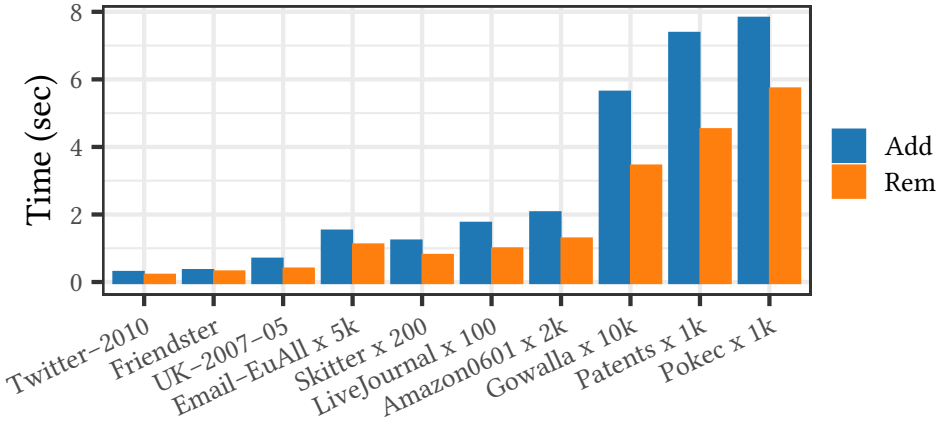
Skitter as the number of nodes changes. We configure half of the cluster to be Streamers to send the graph. The performance is above 2 million edges per second per Agent and scales well.

We next show how important having a *dynamic* system can be. For snapshot-based systems or partially dynamic systems, such as GraphX, the standard approach is to initialize the iterative algorithm with prior outputs, re-initialize any new or changed vertices, and run the iterative algorithm to convergence. If the new graph differs only slightly from the prior graph, the number of iterations may be small, resulting in significant savings over recomputing from scratch. This is the model used by many of the extensions to make GraphX dynamic, e.g., Sprouter [2] and EdgeScaler [204]. A fully dynamic system, on the other hand, can quickly change parts of the graph and only compute with vertices that are active, closer to an asynchronous model. As our dynamic baseline, we *completely ignore partitioning costs* in GraphX and instead use the strategy above, re-initializing only changed vertices and running WCC. By ignoring partitioning costs, we show the best achievable performance if a perfect elastic load balancer is put into GraphX. In Figure 9.15, we show ElGA inserting 100 batches into Twitter-2010. Even on single edge changes, our GraphX baseline never took less than 49.45 seconds. Given ElGA's minimum, average, and maximum runtimes of 0.025, 0.12, and 0.59 seconds on single edge changes, we achieve speedups between 83× to 1962×.

It is important to be able to scale both up and down as the graph size and rate of changes varies. Our architecture is designed to support this. In Figure 9.16 we show the ratio of edges that moved across ElGA when an Agent joins and a random one leaves for each graph. This shows that ElGA can elastically scale as needed without incurring significant overheads. In Figure 9.17, PageRank runs for five iterations on Gowalla starting with 16 nodes. After one iteration, an operator manually scales the cluster from 16 to 64 nodes. ElGA elastically scales and continues the computation, improving the overall runtime. Finally, after the computation is over, the cluster is reduced back, providing cost savings.



(a) The percent of edges moved when adding and then removing an Agent



(b) The total time to add and then remove a single Agent.

Figure 9.16: The cost of adding and removing one Agent, starting from 2048. Multiple Agent changes amortize the cost.

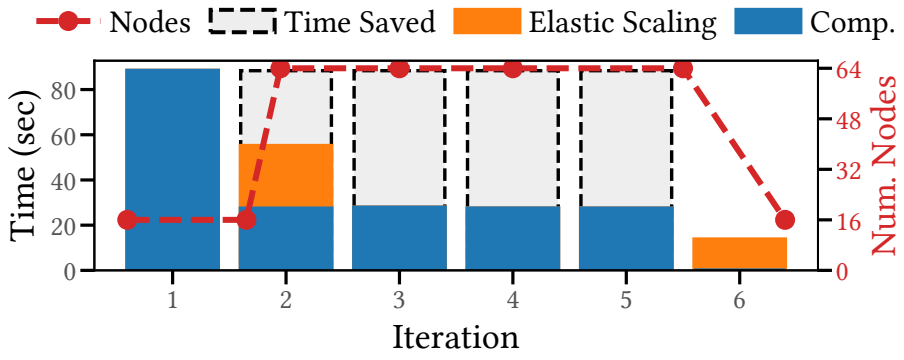


Figure 9.17: PageRank running on Gowalla, manually scaled to 64 nodes during computation and then back to 16.

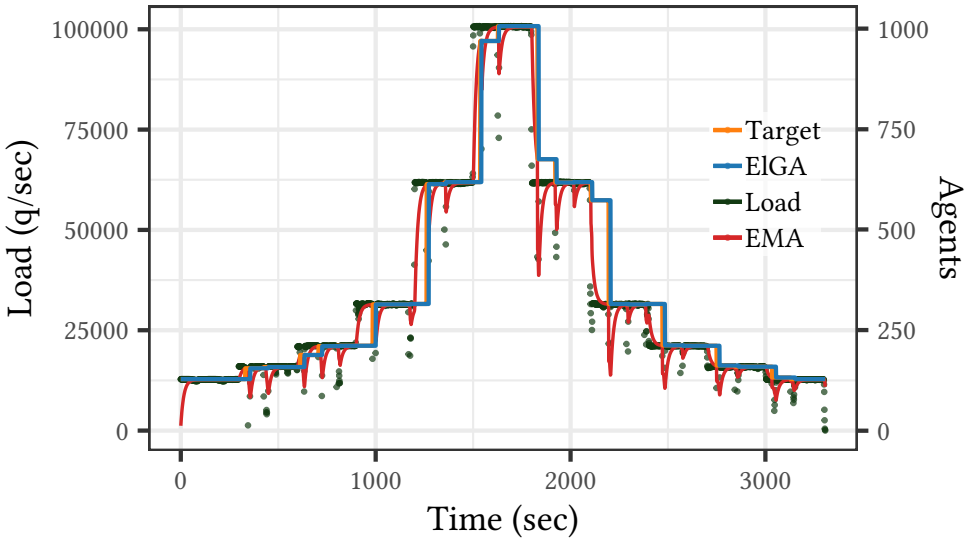


Figure 9.18: Fully elastic autoscaling in ElGA. ElGA converges quickly to match the autoscaling target (Target).

We next evaluate ElGA's elasticity. We implemented a reactive autoscaler that takes a fraction of the exponential moving average (EMA) of 30 seconds of client PageRank vertex query rates to determine the target Agent count. It then waits for 60 seconds before potentially scaling again to allow the EMA to stabilize. Any suitable autoscaler or scaling measures can be used [124]. We varied client request rates with a step function to emulate sudden workload changes on Skitter. The results are shown in Figure 9.18. ElGA quickly converges to the autoscaler's target, evidenced by the mostly overlapping blue and orange lines, and hence elastically matches the load.

9.4 Temporal Support in ElGA

To enable temporal applications in ElGA, we added support for saving timestamps alongside edges. When this option is enabled (`CONFIG_TEMPORAL`), an edge becomes a three-tuple (a, b, t), where t is a 64-bit timestamp. In ElGA without temporal support, edges are stored first in an Abseil flat hash map, and second in a vector of vertices representing the edge endpoints. There is a separate vector for in-edges and out-edges. The edges are not sorted.

With temporal support, there are two additional vectors which hold timestamps and the position in the vector corresponds to the position of the vertex endpoint. Any time an edge (specifically, an *edge change*) is sent across the network, it needs to contain the timestamp of the edge. When an algorithm wishes to use the timestamp of an edge, then it simply accesses the in-times or out-times vector that corresponds to the neighbor, appropriately.

Note that the edges are not re-indexed based on time, which may be preferred for some applications. That means that performing a time-specific query, for example finding edges within a time range, requires an iteration over the edge list. This is similar to any algorithm in EIGA that requires finding a specific vertex as a neighbor, as it needs to traverse the edge list. Some systems will develop a specialized radix tree to support time, e.g., *C*-trees [59], which could be applied as an optimization in EIGA depending on application needs.

9.5 EIGA's Programming Interface

In this section we describe how to develop an application for use inside of EIGA.

EIGA was designed to support a variety of applications, however, EIGA needs to be compiled for a specific algorithm. It further does not support multitenancy, authentication, concurrent algorithm execution, attributes for vertices and edges, non-integral vertices, or multiple graph namespaces. If EIGA is placed into a production environment it would like be desirable for these to be engineered in.

9.5.1 Algorithm Structure

To create a new algorithm, first the build environment needs to be modified to pass in the appropriate compile-time flag. For example, to run weakly connected components, the flag `CONFIG_WCC` is set. This flag is used to indicate which algorithm file to include, e.g., inside of `algorithm.hpp` the `CONFIG_WCC` flag includes the `wccalgorithm.hpp` file. This structure needs to be replicated for any new algorithm.

An algorithm has a single main function that is called for each vertex, following the

```
void init() { id = v; } // Initialize the local vertex state
void run() { // Process as a vertex
    auto old_id = id;
    for (auto r : replica_msgs) id = min(id, r);
    for (auto n : neighbor_msgs) id = min(id, n);
    if (old_id != id) broadcast_reps(id);
    if (old_id != id) broadcast_neighs(id);
    state = INACTIVE;
}
void set_active(msg) { // Runs when receiving message
    if (msg < id) state = ACTIVE;
}
```

Figure 9.19: A high-level example of the weakly connected components algorithm within EIGA.

vertex-centric model. There are a few boilerplate requirements around the main function, and one important decision for the algorithm: what *model* it follows. EIGA supports a full BSP model, an active (lightweight) BSP model, and a full (personalized) model are supported. The algorithm decides this, again at compile time, by setting `#define CONFIG_BSP`, `#define CONFIG_LBSP`, or `#define CONFIG_FULL`. This determines how the algorithm's main run function (that is called per-vertex) will be executed. In BSP, it will be executed each time. In the lightweight BSP, it will be executed if it is active, or if its neighbor is active, and otherwise it will not be executed for a given vertex. In the full model, it will be executed if its requirements are met (it is not waiting on any specific messages from neighbors and it is active).

Any additional storage information used by the algorithm is set in its `VertexStorage` class, which needs to be defined as part of the algorithm's boilerplate. The neighbor and replica messages also need to be set as classes in the boilerplate. This allows an algorithm to specify exactly what it will be sending.

9.5.2 Vertex-Centric Function and Examples

The main function is called `run`. This function largely follows the standard vertex-

```
void init(d=infinity) { dist = d; } // Initialize
void run() { // Process as a vertex
    auto old_dist = dist;
    for (auto r : replica_msgs) dist = min(dist, r);
    for (auto n : neighbor_msgs) dist = min(dist, n);
    if (old_dist != dist) broadcast_reps(dist);
    if (old_dist != dist) broadcast_neighs(dist+1);
    state = INACTIVE;
}
void set_active(msg) { // Runs when receiving message
    if (msg < dist) state = ACTIVE;
}
```

Figure 9.20: A high-level example of the breadth first search algorithm within EIGA.

centric model. For example in weakly connected components, the value `cc`, which stands for connected component, is added to its `VertexStorage`. The high-level example code is shown in Figure 9.19. All of the replica and neighbor messages also have a `cc` value. Then, the `run` code simply reads through each replica and neighbor value, and at the end decides to send out a message if its value changed based on the input values. The mechanism for sending a message is to set the output parameter `notify_X`, where `X` may be in- or out-neighbors or replicas. In the case of weakly connected components, it sends messages to replicas and neighbors at the same time.

In BFS, the `run` function acts similar to weakly connected components, but checks whether the *distance* is lower and if so it remains active and broadcasts. The distance information is part of the `VertexStorage` and included in all messages, as part of the BFS algorithm boilerplate. A high-level example is shown in Figure 9.20. The only difference, besides naming, between Figure 9.19 and Figure 9.20 is the initial values (in which BFS supports a specific node being set to a value of 0, and all others default to ∞) and that the messages sent out to neighbors *increase* the distance relative to the current vertex.

Note that algorithms can choose to only use in- or out-edges, which is not shown here for clarity.

In label propagation, the broadcast occurs only if the most frequent *label* that was re-

ceived differs from the previous labels. Again, the label is included in `VertexStorage` and neighbor and replica messages.

9.5.3 Lightweight BSP and the Full Model

In the case of lightweight BSP and the full model, the algorithm can set its state to `INACTIVE` if it does not want to continue computation the next round. If all vertices are inactive, then the computation will terminate.

For PageRank, the algorithm model differs from the previous examples, as it expects the regular BSP model. This results in a few internal optimizations in ElGA as there is no need to keep track of which vertices are active. Instead, only if all vertices state they should be inactive will anything happen, namely the computation will terminate. In PageRank the `VertexStorage` and messages all contain either a `float` or `double`, specified at compile-time, to hold the current PageRank value. The `run` method simply adds up neighbor values, shares the partial result with replicas, and then normalizes to achieve the correct PageRank value.

In the case of the full model, the algorithm needs to be more precise about which messages it is sending and which messages it requires before becoming active again. Any message that is going to allow it to become active again needs to be set in the `vn_wait` data structure. This is a map of maps, that indicates the vertex is waiting on a specific message (based on a key) from a specific neighbor (the next key in the maps).

9.6 Summary

As graphs continue to grow, many dynamic and large-scale graph algorithms, together with distributed graph processing systems, have been developed. The majority of these efforts, however, fail to consider the high variance from both the distributed graph's natural rate of change and the inherent variance in dynamic computations. We present ElGA, an elastic and scalable dynamic graph analysis system. ElGA performs computation as the underly-

ing graph changes, and can scale as computational demands change. It accomplishes this by combining a shared-nothing architecture with consistent hashing and, to handle heavily skewed graphs, count-min sketches. We show that EIGA outperforms state-of-the-art graph processing systems in terms of runtime on both static and dynamic graph algorithms over variety of real world graphs and their scaled-up replicas.

CHAPTER 10
CONCLUSION AND FUTURE DIRECTIONS

10.1 Conclusion

In this dissertation, we focus on finding dense regions of massively changing graphs. Our contributions are in three main areas.

First, we extend algorithms and theories surrounding dense regions on changing graphs in three parts: unifying nuclei to cores, maintaining cores, and providing a new temporal dense region called core chains. We provide the first dynamic maintenance algorithms for *nuclei*, a computable yet effective dense region target. We do so by unifying all nuclei into a standard target, k-cores, through the construction and use of a specific hypergraph. Our nuclei maintenance algorithm is faster—over 90×—than specialized hand-crafted maintenance algorithms for trusses, a specific type of nuclei. We then demonstrate that the k-core dynamic graph approaches to date have largely focused on the *density levels* for vertices, not on the dense regions themselves. We devise an index to maintain cores themselves for both graphs and hypergraphs, which through our unifying framework apply to nuclei. Our index supports dynamic graph batch updates through the use of a directed acyclic graph made out of *subcores*. From this acyclic graph, we build and maintain the tree of dense regions, capturing the hierarchy. Both of these algorithms are dynamic maintenance algorithms, and so they return the most current output at any point in time. We then define a new *temporal dense region*, which we call core chains, that link together regions of a dense nuclei hierarchy over time. We show that core chains built with higher order nuclei are able to effectively identify meaningful regions of changing graphs. Prior approaches fail to find meaningful regions, whereas we identify ant behavior and research groups.

Second, we address scaling up on shared-memory systems in two parts: system level

improvements through a fast input and output library and parallel algorithms to maintain cores on graphs and hypergraphs. When graphs become medium sized, it takes hours or longer to perform operations on graphs with a single thread. We address a critical systems issue: parallel input and output. We provide the first known parallel graph-specific input and output library, named PIGO. It supports reading edge lists, adjacency lists, and other formats, such as tensors. We show that all known shared-memory graph systems have large end-to-end runtime improvements with PIGO, up to 38×. We then develop the *first scalable and parallel* k-core and hypergraph k-core batch-dynamic algorithms. We provide two algorithms: one based on incrementing a large region, and then running h-indices iteratively to correct for over-incrementing, and the other that propagates a graph change while iteratively computing h-indices. Both algorithms scale well with insertions, deletions, and mixed batches, achieving up to 13× speedup with 16 cores and more than 4× faster than prior algorithms. Together, these bring effective and useful dense region tracking to medium sized graphs that fit in shared-memory systems.

Third, we scale out to support distributed-memory systems in two parts: a parallel h-index algorithm that enables core maintenance on distributed systems and an elastic, dynamic, and scalable distributed-memory graph system. To bring dense region tracking to distributed-memory systems, we develop the *first parallel h-index* algorithm. We build on the recursive nature of the problem and, taking linear passes with a continually reducing subset of data, we find the h-index in parallel. This enables our shared-memory k-core and hypergraph k-core approaches to scale outwards, computing h-indices with 3 trillion integers in around 10 seconds. Large graphs require distributed-memory systems: even if a compressed form can fit into a shared-memory system, in real-world deployments there are typically thousands or more of concurrent clients performing queries or graph updates, and furthermore the rate of change itself is highly variable. To efficiently support a highly variable rate, we identify the need for *elasticity* in the graph systems. We develop the first known elastic and dynamic graph system, EIGA. EIGA partitions edges using two layers of

consistent hashing, the first to identify a set of machines that hold all edges for the source vertex, and the second to identify the specific machine that holds the edge in question. The number of machines is set based on a *sketch* of the degree of the vertex, as directly storing all vertex degrees does not scale. We show that ElGA is not only able to support dynamic graphs and elastically scale, but it is also fast: its per-iteration runtimes beat state-of-the-art static system competitors by over 2.4×, and against the more well supported dynamic system, it brings improvements of up to 1962×.

Through these three areas, we bring dense region tracking to massive graphs.

10.2 Future Directions

There are many avenues for further exploration. We identify three exciting future directions here. The first direction is to explore approximate results. In many cases having the exact result is not particularly useful. This is especially true in real-world graphs that are continuously changing, as the real, underlying graph may never be known or may be out-of-date by the time it reaches the graph system. There are many ways that nuclei can be extended to support such approximate goals. One promising idea is to relax the strict clique requirement for inclusion. For example, any approximately dense region could be used as s-cliques, and any approximately dense subgraph inside of s-cliques could replace r-cliques. The connectivity constraint would remain. It is possible that this generalized nucleus approach could provide around same insight as nuclei, but could be computed much quicker, especially for graphs with large cliques.

The second direction is concerned with attributes and labels on graphs. In this work we consider basic graphs: they are undirected, do not have multiedges, and have no weights or attributes on either vertices or edges. This is a common starting place for graph analytics. However, in many cases vertices and edges can be augmented with additional data, such as direction and attributes, and multiple edges can simultaneously exist. This reflects real-world data better. An important problem in graph analytics is to uncover techniques to use

this additional data to further improve the quality of results. There is a lot of potential in a slightly extended graph framework, where vertices can exist on a spectrum (e.g., either red, or blue, or somewhere in-between). This extended framework has natural applications in environments such as disinformation and computer security. Understanding what a dense region is in this space, and determining how to effectively compute and use them, is an interesting and valuable problem.

The third direction is to explore further and deeper temporal behaviors. This is an increasingly important goal in graph analysis. Many graphs change over time, and it is important to develop algorithms that can keep up with the continuous changes to the graph. However, simply keeping up throws away an interesting dimension, time. In this work we developed one effective temporal dense region, core chains. However, temporal analysis in general remains a largely untouched space, with numerous opportunities throughout. Exciting future work includes systems work to develop different strategies for querying historical edges, algorithms work to develop further strategies for understanding graphs as they change over time, and extending both approximate graph algorithms and attributed graphs to the temporal setting.

REFERENCES

[1] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: An experimental study," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1590–1603, 2018.

[2] T. Abughofa and F. Zulkernine, "Sprouter: Dynamic graph processing over data streams at scale," in *International Conference on Database and Expert Systems Applications*, Springer, 2018, pp. 321–328.

[3] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala, "Parallel batch-dynamic graph connectivity," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 381–392.

[4] C. C. Aggarwal, Y. Li, P. S. Yu, and R. Jin, "On dense pattern mining in graph streams," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 975–984, 2010.

[5] K. J. Ahn, S. Guha, and A. McGregor, "Graph sketches: Sparsification, spanners, and subgraphs," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, 2012, pp. 5–14.

[6] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and Ö. Ulusoy, "Distributed k-Core View Materialization and Maintenance for Large Dynamic Graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 10, pp. 2439–2452, 2014.

[7] S. Alonso, F. J. Cabrerizo, E. Herrera-Viedma, and F. Herrera, "H-index: A review focused in its variants, computation and standardization for different scientific fields," *Journal of Informetrics*, vol. 3, no. 4, pp. 273–289, 2009.

[8] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck, "Incremental Evaluation of Computational Circuits," in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '90, San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1990, pp. 32–42.

[9] I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "K-core decomposition: A tool for the analysis of large scale internet graphs," *arXiv preprint cs.NI/0511007*, 2005.

[10] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Advances in neural information processing systems*, 2006, pp. 41–50.

[11] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, "Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows," *Proceedings of the VLDB Endowment*, vol. 11, no. 6, pp. 691–704, 2018.

[12] K. Ammar and M. T. Özsu, "Experimental analysis of distributed graph systems," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1151–1164, 2018.

[13] A. Angel, N. Sarkas, N. Koudas, and D. Srivastava, "Dense subgraph maintenance under streaming edge weight updates for real-time story identification," *Proceedings of the VLDB Endowment*, vol. 5, no. 6, pp. 574–585, 2012.

[14] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis, "Distributed k-core decomposition and maintenance in large dynamic graphs," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 161–168.

[15] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[16] S. Asur, S. Parthasarathy, and D. Ucar, "An Event-Based Framework for Characterizing the Evolutionary Behavior of Interaction Graphs," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '07*, ser. KDD '07, San Jose, California, USA: ACM Press, 2007, pp. 913–921.

[17] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 44–54.

[18] W. Bai, Y. Chen, and D. Wu, "Efficient temporal core maintenance of massive graphs," *Information Sciences*, vol. 513, pp. 324–340, 2020.

[19] W. Bai, Y. Zhang, X. Liu, M. Chen, and D. Wu, "Efficient Core Maintenance of Dynamic Graphs," in *International Conference on Database Systems for Advanced Applications*, Springer, 2020, pp. 658–665.

[20] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, "Efficient and effective community search," *Data mining and knowledge discovery*, vol. 29, no. 5, pp. 1406–1433, 2015.

[21] J. A. Barnes, "Graph Theory and Social Networks: A Technical Comment on Connectedness and Connectivity," *Sociology*, vol. 3, no. 2, pp. 215–232, 1969.

[22] S. Béal, S. Ferrières, E. Rémila, and P. Solal, "An axiomatization of the iterated h-index and applications to sport rankings," 2016.

[23] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," 2015.

[24] C. Belth, X. Zheng, and D. Koutra, "Mining persistent activity in continually evolving networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 934–944.

[25] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *SIGACT News*, vol. 12, no. 3, pp. 36–44, 1980.

[26] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems," 2019.

[27] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries," 2019.

[28] K. Bhawalkar, J. Kleinberg, K. Lewi, T. Roughgarden, and A. Sharma, "Preventing unraveling in social networks: The anchored k-core problem," *SIAM Journal on Discrete Mathematics*, vol. 29, no. 3, pp. 1452–1475, 2015.

[29] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, P10008, 2008.

[30] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, 1973.

[31] P. Bogdanov, M. Mongiovi, and A. K. Singh, "Mining Heavy Subgraphs in Time-Evolving Networks," in *2011 IEEE 11th International Conference on Data Mining*, IEEE, 2011, pp. 81–90.

[32] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *Proceedings of the 13th Conference on World Wide Web*, Manhattan, USA: ACM Press, 2004, pp. 595–601.

[33] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[34] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks," in *Proceedings of the 20th International Conference on World wide web*, S. Srinivasan,

K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds., ACM Press, 2011, pp. 587–596.

[35] L. Bornmann, R. Mutz, S. E. Hug, and H.-D. Daniel, "A multilevel meta-analysis of studies reporting correlations between the h index and 37 different h index variants," *Journal of Informetrics*, vol. 5, no. 3, pp. 346–359, 2011.

[36] D. Borthakur *et al.*, "Hdfs architecture guide," *Hadoop Apache Project*, vol. 53, no. 1-13, p. 2, 2008.

[37] U. Brandes, P. Kenis, J. Lerner, and D. van Raaij, "Network analysis of collaboration structure in Wikipedia," in *Proceedings of the 18th International Conference on World wide web - WWW 09*, ACM Press, 2009, pp. 731–740.

[38] A. Z. Broder, A. M. Frieze, and E. Upfal, "On the satisfiability and maximum satisfiability of random 3-cnf formulas.," in *SODA*, vol. 93, 1993, pp. 322–330.

[39] M. Brunato, H. H. Hoos, and R. Battiti, "On Effectively Finding Maximal Quasi-cliques in Graphs," in *International Conference on learning and intelligent optimization*, Springer, 2008, pp. 41–55.

[40] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, K. Kalyanaraman, R. Khanna, J. Pao, *et al.*, "A1: A distributed in-memory graph database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 329–344.

[41] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[42] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, SIAM, 2004, pp. 442–446.

[43] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*, Springer, 2002, pp. 693–703.

[44] N. Chatterjee and S. Sinha, "Understanding the mind of a worm: Hierarchical network structure underlying nervous system function in c. elegans," *Progress in brain research*, vol. 168, pp. 145–153, 2007.

[45] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: An efficient data structure for static support lookup tables," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, Citeseer, 2004, pp. 30–39.

[46] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 85–98.

[47] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and mobility: User movement in location-based social networks," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011, pp. 1082–1090.

[48] L. Chu, Y. Zhang, Y. Yang, L. Wang, and J. Pei, "Online density bursting subgraph detection from temporal graphs," *Proceedings of the VLDB Endowment*, vol. 12, no. 13, pp. 2353–2365, 2019.

[49] M. Ciaperoni, E. Galimberti, F. Bonchi, C. Cattuto, F. Gullo, and A. Barrat, "Relevance of temporal cores for epidemic spread in temporal networks," *Scientific reports*, vol. 10, no. 1, pp. 1–15, 2020.

[50] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, pp. 3–1, 2008.

[51] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[52] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, ACM, 2013, pp. 277–288.

[53] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, 2014, pp. 991–1002.

[54] J. Dai, Y. Li, X. Fan, J. Sun, and Y. Zhao, "Finding Early Bursting Cohesive Subgraphs in Large Temporal Networks," in *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, IEEE, Oct. 2021.

[55] A. Das, M. Svendsen, and S. Tirthapura, "Incremental maintenance of maximal cliques in a dynamic graph," *The VLDB Journal*, vol. 28, no. 3, pp. 351–375, 2019.

[56] N. S. Dasari, R. Desh, and M. Zubair, "ParK: An efficient algorithm for k-core decomposition on multicore processors," in *2014 IEEE International Conference on Big Data (Big Data)*, IEEE, IEEE, Oct. 2014, pp. 9–16.

[57] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[58] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017, pp. 293–304.

[59] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: ACM, Jun. 2019, pp. 918–934.

[60] L. Dhulipala, D. Durfee, J. Kulkarni, R. Peng, S. Sawlani, and X. Sun, "Parallel batch-dynamic graphs: Algorithms and lower bounds," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2020, pp. 1300–1319.

[61] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2017.

[62] D. Ding, H. Li, Z. Huang, and N. Mamoulis, "Efficient fault-tolerant group recommendation using alpha-beta-core," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 2047–2050.

[63] J. R. Douceur, "The sybil attack," in *International Workshop on Peer-to-Peer Systems*, Springer, 2002, pp. 251–260.

[64] Y. Dourisboure, F. Geraci, and M. Pellegrini, "Extraction and classification of dense communities in the web," in *Proceedings of the 16th International Conference on World Wide Web*, ACM, 2007, pp. 461–470.

[65] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[66] ECMA, "Data interchange on 12,7 mm 48-track magnetic tape cartridges - DLT1 format," Tech. Rep., 1993.

[67] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*, IEEE, 2012, pp. 1–5.

[68] L. Egghe and R. Rousseau, "A geometric relation between the h-index and the lorenz curve," *Scientometrics*, vol. 119, no. 2, pp. 1281–1284, 2019.

[69] ——, "Solution by step functions of a minimum problem in L2[0,T], using generalized h- and g-indices," *Journal of Informetrics*, vol. 13, no. 3, pp. 785–792, Aug. 2019.

[70] ——, "The h-index formalism," *Scientometrics*, vol. 126, no. 7, pp. 6137–6145, 2021.

[71] M. Eidsaa and E. Almaas, "*s*-core network decomposition: A generalization of *k*-core analysis to weighted networks," *Phys. Rev. E*, vol. 88, p. 062 819, 6 Dec. 2013.

[72] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, "Sparsification—a technique for speeding up dynamic graph algorithms," *Journal of the ACM (JACM)*, vol. 44, no. 5, pp. 669–696, Sep. 1997.

[73] B. Erb, D. Meißner, F. Kargl, B. A. Steer, F. Cuadrado, D. Margan, and P. Pietzuch, "Graphtides: A framework for evaluating stream-based graph processing platforms," in *Proceedings of the 1st ACM SIGMOD joint International Workshop on graph data management experiences & systems (GRADES) and network data analytics (NDA)*, 2018, pp. 1–10.

[74] B. Erb, D. Meissner, J. Pietron, and F. Kargl, "Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, 2017, pp. 78–87.

[75] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, "The ldbc social network benchmark: Interactive workload," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 619–630.

[76] W. Fan, C. Hu, M. Liu, P. Lu, Q. Yin, and J. Zhou, "Dynamic scaling for parallel graph computations," *Proceedings of the VLDB Endowment*, vol. 12, no. 8, pp. 877–890, 2019.

[77] W. Fan, C. Hu, and C. Tian, "Incremental graph computations: Doable and undoable," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ACM, 2017, pp. 155–169.

[78] W. Fan, P. Lu, W. Yu, J. Xu, Q. Yin, X. Luo, J. Zhou, and R. Jin, "Adaptive asynchronous parallelization of graph algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 45, no. 2, pp. 1–45, 2020.

[79] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu, "Parallelizing sequential graph computations," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 4, pp. 1–39, 2018.

[80] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu, "Effective and efficient attributed community search," *The VLDB Journal*, vol. 26, no. 6, pp. 803–828, 2017.

[81] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *The VLDB Journal*, vol. 29, no. 1, pp. 353–392, 2020.

[82] Y. Fang, Y. Yang, W. Zhang, X. Lin, and X. Cao, "Effective and efficient community search over large heterogeneous information networks," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, pp. 854–867, 2020.

[83] G. Feng, Z. Ma, D. Li, X. Zhu, Y. Cai, W. Han, and W. Chen, "Risgraph: A real-time streaming system for evolving graphs," *arXiv preprint arXiv:2004.00803*, 2020.

[84] A. Fidel, N. M. Amato, L. Rauchwerger, *et al.*, "Kla: A new algorithmic paradigm for parallel graph computations," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, IEEE, 2014, pp. 27–38.

[85] H. A. Filho, J. Machicao, and O. M. Bruno, "A hierarchical model of metabolic machinery based on the k core decomposition of plant metabolic networks," *PloS one*, vol. 13, no. 5, e0195843, 2018.

[86] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, Feb. 2010.

[87] S. Fortunato and D. Hric, "Community detection in networks: A user guide," *Physics Reports*, vol. 659, pp. 1–44, Nov. 2016.

[88] E. Fratkin, B. T. Naughton, D. L. Brutlag, and S. Batzoglou, "MotifCut: Regulatory motifs finding with maximum density subgraphs," *Bioinformatics*, vol. 22, no. 14, e150–e157, 2006.

[89] K. Gabert and Ü. V. Çatalyürek, "PIGO: A Parallel Graph Input/Output Library," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Workshop on Graphs, Architectures, Programming, and Learning (GrAPL)*, IEEE, May 2021.

[90] K. Gabert, A. Pınar, and Ü. V. Çatalyürek, "A Unifying Framework to Identify Dense Subgraphs on Streams: Graph Nuclei to Hypergraph Cores," in *Proceedings of the 14th ACM International Conference on Web Search and Data Mining (WSDM)*, ser. WSDM 21, Virtual Event, Israel: ACM, Mar. 2021, pp. 689–697.

[91] ——, "Shared-Memory Scalable k-Core Maintenance on Dynamic Graphs and Hypergraphs," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Workshop on Parallel and Distributed Processing for Computational Social Systems (ParSocial)*, IEEE, May 2021, pp. 998–1007.

[92] ——, "Batch Dynamic Algorithm to Find k-Cores and Hierarchies," ArXiv, Tech. Rep. arXiv:2203.13095, Mar. 2022.

[93] K. Gabert, K. Sancak, M. Y. Özkaya, A. Pınar, and Ü. V. Çatalyürek, "ElGA: Elastic and Scalable Dynamic Graph Analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, Nov. 2021.

[94] K. G. Gabert and A. Pinar, "Extracting Stable Community Information on Relational Data," in *Conference on Data Analysis (CoDA)*, Feb. 2020.

[95] E. Galimberti, A. Barrat, F. Bonchi, C. Cattuto, and F. Gullo, "Mining (maximal) Span-cores from Temporal Networks," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, 2018, pp. 107–116.

[96] E. Galimberti, F. Bonchi, F. Gullo, and T. Lanciano, "Core Decomposition in Multilayer Networks, Theory, algorithms, and applications," *ACM Transactions on Knowledge Discovery from Data*, vol. 14, no. 1, pp. 1–40, Feb. 2020.

[97] E. Galimberti, M. Ciaperoni, A. Barrat, F. Bonchi, C. Cattuto, and F. Gullo, "Span-Core Decomposition for Temporal Networks: Algorithms and Applications," *ACM Trans. Knowl. Discov. Data*, vol. 15, no. 1, Dec. 2020.

[98] L. Gao, G. Gao, D. Ma, and L. Xu, "Coreness variation rule and fast updating algorithm for dynamic networks," *Symmetry*, vol. 11, no. 4, p. 477, 2019.

[99] J. García-Algarra, J. M. Pastor, J. M. Iriondo, and J. Galeano, "Ranking of critical species to preserve the functionality of mutualistic networks using the k-core decomposition," *PeerJ*, vol. 5, e3321, 2017.

[100] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1012–1023, 2013.

[101] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2013.

[102] P. B. Gibbons and Y. Matias, "Synopsis data structures for massive data sets," *External memory algorithms*, vol. 50, pp. 39–70, 1999.

[103] D. Gibson, R. Kumar, and A. Tomkins, "Discovering Large Dense Subgraphs in Massive Graphs," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, Trondheim, Norway: VLDB Endowment, 2005, pp. 721–732.

[104] A. V. Goldberg, *Finding a maximum density subgraph*. University of California Berkeley, 1984.

[105] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 599–613.

[106] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, IEEE, Sep. 2011, pp. 792–799.

[107] *Google programming contest*, 2002.

[108] D. Greene, D. Doyle, and P. Cunningham, "Tracking the Evolution of Communities in Dynamic Social Networks," in *2010 International Conference on Advances in Social Networks Analysis and Mining*, IEEE, Aug. 2010, pp. 176–183.

[109] P. Hagmann, L. Cammoun, X. Gigandet, R. Meuli, C. J. Honey, V. J. Wedeen, and O. Sporns, "Mapping the structural core of human cerebral cortex," *PLoS Biol*, vol. 6, no. 7, e159, 2008.

[110] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[111] S. Heidari and R. Buyya, "A Cost-Efficient Auto-Scaling Algorithm for Large-Scale Graph Processing in Cloud Environments with Heterogeneous Resources," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1729–1741, Aug. 2019.

[112] S. Heidari, R. N. Calheiros, and R. Buyya, "Igiraph: A cost-efficient framework for processing large-scale graphs on public clouds," in *2016 16th IEEE/ACM Interna-*

tional Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, 2016, pp. 301–310.

[113] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–53, 2018.

[114] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 23–27.

[115] P. Hintjens, *ZeroMQ: messaging for many applications*. 2013.

[116] J. E. Hirsch, "An index to quantify an individual's scientific research output," *Proceedings of the National Academy of Sciences*, vol. 102, no. 46, pp. 16 569–16 572, 2005.

[117] G. Ho, A. Cidon, L. Gavish, M. Schweighauser, V. Paxson, S. Savage, G. M. Voelker, and D. Wagner, "Detecting and characterizing lateral phishing at scale," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1273–1290.

[118] P. Holme and J. Saramäki, "Temporal networks," *Physics Reports*, vol. 519, no. 3, pp. 97–125, Oct. 2012.

[119] J. Hopcroft, O. Khan, B. Kulis, and B. Selman, "Tracking evolving communities in large linked networks," *Proceedings of the National Academy of Sciences*, vol. 101, no. suppl 1, pp. 5249–5253, Apr. 2004.

[120] R. Hovden, "Bibliometrics for Internet media: Applying the h-index to You Tube," *Journal of the American Society for Information Science and Technology*, vol. 64, no. 11, pp. 2326–2331, 2013.

[121] Q.-S. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen, "Faster parallel core maintenance algorithms in dynamic graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1287–1300, 2019.

[122] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, ACM, 2014, pp. 1311–1322.

[123] W.-C. Hung and C.-Y. Tseng, "Maximum (L, K)-Lasting Cores in Temporal Social Networks," in *International Conference on Database Systems for Advanced Applications*, Springer, 2021, pp. 336–352.

[124] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 75–86.

[125] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, I. G. Tănase, Y. Xia, L. Nai, and P. Boncz, "LDBC graphalytics, A benchmark for large-scale graph analysis on parallel and distributed platforms," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1317–1328, Sep. 2016.

[126] A. Iyer, L. E. Li, and I. Stoica, "Celliq: Real-time cellular network analytics at scale," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 309–322.

[127] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, 2016, pp. 1–6.

[128] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, "ASAP: Fast, approximate graph pattern mining at scale," in *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*, 2018, pp. 745–761.

[129] A. P. Iyer, A. Panda, M. Chowdhury, A. Akella, S. Shenker, and I. Stoica, "Monarch: Gaining command on geo-distributed graph analytics," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[130] V. Jethava and N. Beerenwinkel, "Finding Dense Subgraphs in Relational Graphs," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2015, pp. 641–654.

[131] H. Jin, N. Wang, D. Yu, Q.-S. Hua, X. Shi, and X. Xie, "Core maintenance in dynamic graphs: A parallel approach based on matching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2416–2428, 2018.

[132] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: A high-compression indexing scheme for reachability query," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ACM, 2009, pp. 813–826.

[133] M. Jung, "Openexpress: Fully hardware automated open research framework for future fast nvme devices," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USA: USENIX Association, 2020.

[134] M. F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal distributed hash table," in *International Workshop on Peer-to-Peer Systems*, Springer, 2003, pp. 98–107.

[135] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2017, pp. 1482–1491.

[136] B. Kahveci and B. Gedik, "Joker: Elastic stream processing with organic adaptation," *Journal of Parallel and Distributed Computing*, vol. 137, pp. 205–223, 2020.

[137] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 654–663.

[138] R. M. Karp, "Reducibility among Combinatorial Problems," in *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[139] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2019, pp. 1–7.

[140] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica, "Zipg: A memory-efficient graph store for interactive queries," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1149–1164.

[141] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, "Identification of influential spreaders in complex networks," *Nature physics*, vol. 6, no. 11, pp. 888–893, 2010.

[142] T. Klingberg and R. Manfredi, *Gnutella 0.6*, 2002.

[143] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The SuiteSparse matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019.

[144] Y.-X. Kong, G.-Y. Shi, R.-J. Wu, and Y.-C. Zhang, "K-core: Theories and applications," *Physics Reports*, vol. 832, pp. 1–32, 2019.

[145] V. Kostakos, "Temporal graphs," *Physica A: Statistical Mechanics and its Applications*, vol. 388, no. 6, pp. 1007–1023, 2009.

[146] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 249–263.

[147] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tompkins, and E. Upfal, "The web as a graph," in *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2000, pp. 1–10.

[148] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "Trawling the Web for emerging cyber-communities," *Computer networks*, vol. 31, no. 11-16, pp. 1481–1493, 1999.

[149] S. Kumar, B. Hooi, D. Makhija, M. Kumar, C. Faloutsos, and V. Subrahmanian, "Rev2: Fraudulent user prediction in rating platforms," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, ACM, 2018, pp. 333–341.

[150] S. Kumar, F. Spezzano, V. Subrahmanian, and C. Faloutsos, "Edge weight prediction in weighted signed networks," in *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, IEEE, 2016, pp. 221–230.

[151] J. Kunegis, "KONECT, The koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web Companion*, ACM, ACM Press, 2013, pp. 1343–1350.

[152] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" In *Proceedings of the 19th International Conference on World wide web*, ACM Press, 2010, pp. 591–600.

[153] M. Lahiri and T. Y. Berger-Wolf, "Mining Periodic Behavior in Dynamic Social Networks," in *2008 Eighth IEEE International Conference on Data Mining*, IEEE, 2008, pp. 373–382.

[154] M. M. Lee, I. Roy, A. AuYoung, V. Talwar, K. Jayaram, and Y. Zhou, "Views and transactional storage for large graphs," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, Springer, 2013, pp. 287–306.

[155] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal, "A survey of algorithms for dense subgraph discovery," in *Managing and Mining Graph Data*, Springer, 2010, pp. 303–336.

[156] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, 5–es, 2007.

[157] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 631–636.

[158] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2005, pp. 177–187.

[159] ——, "Graph evolution: Densification and shrinking diameters," *ACM transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, 2–es, 2007.

[160] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, http://snap.stanford.edu/data, Jun. 2014.

[161] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[162] M. Ley, "The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives," in *String Processing and Information Retrieval*, Springer, Springer Berlin Heidelberg, 2002, pp. 1–10.

[163] R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai, "Persistent Community Search in Temporal Networks," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, IEEE, 2018, pp. 797–808.

[164] R.-H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 10, pp. 2453–2465, 2013.

[165] Y. Li, J. Liu, H. Zhao, J. Sun, Y. Zhao, and G. Wang, "Efficient continual cohesive subgraph search in large temporal graphs," *World Wide Web*, vol. 24, no. 5, pp. 1483–1509, 2021.

[166] L. Lin, P. Yuan, R.-H. Li, J. Wang, L. Liu, and H. Jin, "Mining Stable Quasi-Cliques on Temporal Networks," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–15, 2021.

[167] L. Lin, P. Yuan, R. Li, and H. Jin, "Mining Diversified Top-r Lasting Cohesive Subgraphs on Temporal Networks," *IEEE Transactions on Big Data*, pp. 1–1, 2021.

[168] Y.-R. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng, "FacetNet: A Framework for Analyzing Communities and Their Evolutions in Dynamic Networks," in *Pro-*

ceeding of the 17th International Conference on World Wide Web, New York, NY, USA: ACM Press, 2008, pp. 685–694.

[169] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian, "Hierarchical core maintenance on large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 757–770, 2021.

[170] Q. Linghu, F. Zhang, X. Lin, W. Zhang, and Y. Zhang, "Global reinforcement of social networks: The anchored coreness problem," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2211–2226.

[171] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation: An index-based approach," in *The World Wide Web Conference*, 2019, pp. 1130–1141.

[172] X. Liu, T. Ge, and Y. Wu, "Finding densest lasting subgraphs in dynamic graphs: A stochastic approach," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, IEEE, 2019, pp. 782–793.

[173] ——, "A Stochastic Approach to Finding Densest Temporal Subgraphs in Dynamic Graphs," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2020.

[174] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley, "The H-index of a network node and its relation to degree and coreness," *Nature Communications*, vol. 7, no. 1, p. 10 168, Jan. 2016.

[175] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Efficient erasure correcting codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 569–584, 2001.

[176] Q. Luo, D. Yu, X. Cheng, Z. Cai, J. Yu, and W. Lv, "Batch Processing for Truss Maintenance in Large Dynamic Graphs," *IEEE Transactions on Computational Social Systems*, 2020.

[177] S. Ma, R. Hu, L. Wang, X. Lin, and J. Huai, "Fast Computation of Dense Temporal Subgraphs," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, IEEE, 2017, pp. 361–372.

[178] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*, IEEE, 2015, pp. 363–374.

[179] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of*

the 2010 ACM SIGMOD International Conference on Management of data, 2010, pp. 135–146.

[180] F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis, "The core decomposition of networks: Theory, algorithms and applications," *The VLDB Journal*, vol. 29, no. 1, pp. 61–92, Jan. 2020.

[181] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proceedings of the Linux symposium*, Citeseer, vol. 2, 2007, pp. 21–33.

[182] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "Lagraph: A community effort to collect graph algorithms built on top of the graphblas," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2019, pp. 276–284.

[183] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 417–427, 1983.

[184] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1–39, 2015.

[185] A. F. McDaid, D. Greene, and N. Hurley, "Normalized mutual information to evaluate overlapping community finding algorithms," *arXiv preprint arXiv:1110.2515*, 2011.

[186] A. McGregor, "Graph stream algorithms, A survey," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 9–20, May 2014.

[187] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what COST?" In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[188] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential Dataflow," in *CIDR*, 2013.

[189] P. Mell, T. Grance, *et al.*, "The NIST definition of cloud computing," 2011.

[190] D. P. Mersch, A. Crespi, and L. Keller, "Tracking Individuals Shows Spatial Fidelity Is a Key Regulator of Ant Social Organization," *Science*, vol. 340, no. 6136, pp. 1090–1093, May 2013.

[191] A. E. Mislove, "Online social networks: Measurement, analysis, and applications to distributed information systems," Ph.D. dissertation, 2009.

[192] M. Mitzenmacher and G. Varghese, "Biff (Bloom filter) codes: Fast error correction for large data sets," in *2012 IEEE International Symposium on Information Theory Proceedings*, IEEE, 2012, pp. 483–487.

[193] R. J. Mokken *et al.*, "Cliques, clubs and clans," *Quality & Quantity*, vol. 13, no. 2, pp. 161–173, 1979.

[194] M. Molloy, "Cores in random hypergraphs and boolean formulas," *Random Structures & Algorithms*, vol. 27, no. 1, pp. 124–135, 2005.

[195] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 145–156.

[196] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE TPDS*, vol. 24, no. 2, pp. 288–300, 2012.

[197] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[198] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 439–455.

[199] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 456–471.

[200] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[201] G. Palla, A.-L. Barabási, and T. Vicsek, "Quantifying social group evolution," *Nature*, vol. 446, no. 7136, pp. 664–667, Apr. 2007.

[202] J. Pardalos and M. Resende, "On maximum clique problems in very large graphs," *DIMACS series*, vol. 50, pp. 119–130, 1999.

[203] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of data*, 1988, pp. 109–116.

[204] D. Presser, F. Siqueira, L. Rodrigues, and P. Romano, "EdgeScaler: effective elastic scaling for graph stream processing systems," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 39–50.

[205] M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell, "Supporting on-demand elasticity in distributed graph processing," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2016, pp. 12–21.

[206] H. Qin, R.-H. Li, G. Wang, L. Qin, Y. Cheng, and Y. Yuan, "Mining Periodic Cliques in Temporal Networks," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, IEEE, 2019, pp. 1130–1141.

[207] H. Qin, R.-H. Li, G. Wang, L. Qin, Y. Yuan, and Z. Zhang, "Mining Bursting Communities in Temporal Graphs," *arXiv preprint arXiv:1911.02780*, 2019.

[208] H. Qin, R. Li, Y. Yuan, G. Wang, W. Yang, and L. Qin, "Periodic Communities Mining in Temporal Networks: Concepts and Algorithms," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2020.

[209] raffi, *New Tweets per second record, and how!* https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how, Accessed June 28, 2021; Archive at https://web.archive.org/web/20210628160850/https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how, 2013.

[210] S. Rajasekaran, "Randomized parallel selection," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer, 1990, pp. 215–224.

[211] G. Ramalingam and T. Reps, "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, no. 1-2, pp. 233–277, 1996.

[212] T. O. Richardson, T. Kay, R. Braunschweig, O. A. Journeau, M. Rüegg, S. McGregor, P. D. L. Rios, and L. Keller, "Ant behavioral maturation is mediated by a stochastic transition between two fundamental states," *Current Biology*, vol. 31, no. 10, 2253–2260.e3, May 2021.

[213] M. Ripeanu, A. Iamnitchi, and I. Foster, "Mapping the gnutella network," *IEEE Internet Computing*, vol. 6, no. 1, p. 50, 2002.

[214] R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *AAAI*, 2015.

[215] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[216] P. Rozenshtein, F. Bonchi, A. Gionis, M. Sozio, and N. Tatti, "Finding events in temporal networks: Segmentation meets densest subgraph discovery," *Knowledge and Information Systems*, vol. 62, no. 4, pp. 1611–1639, Oct. 2020.

[217] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, pp. 420–431, 2017.

[218] S. Sallinen, R. Pearce, and M. Ripeanu, "Incremental graph processing for online analytics," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2019, pp. 1007–1018.

[219] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, *et al.*, "Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2017, pp. 1–6.

[220] S.-V. Sanei-Mehri, A. Das, H. Hashemi, and S. Tirthapura, "Mining Largest Maximal Quasi-Cliques," *ACM Transactions on Knowledge Discovery from Data*, vol. 15, no. 5, pp. 1–21, Jun. 2021.

[221] A. E. Sariyüce and A. Pinar, "Fast hierarchy construction for dense subgraphs," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 97–108, 2016.

[222] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *VLDB*, vol. 12, no. 1, pp. 43–56, 2018.

[223] A. E. Sariyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek, "Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions," in *Proceedings of the 24th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, May 2015, pp. 927–937.

[224] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Incremental k-core decomposition: Algorithms and evaluation," *The VLDB Journal*, vol. 25, no. 3, pp. 425–447, 2016.

[225] ——, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.

[226] A. E. Sarıyüce and A. Pinar, "Peeling bipartite networks for dense subgraph discovery," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 2018, pp. 504–512.

[227] S. B. Seidman and B. L. Foster, "A graph-theoretic generalization of the clique concept," *Journal of Mathematical sociology*, vol. 6, no. 1, pp. 139–154, 1978.

[228] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[229] K. Semertzidis, E. Pitoura, E. Terzi, and P. Tsaparas, "Finding lasting dense subgraphs," *Data Mining and Knowledge Discovery*, vol. 33, no. 5, pp. 1417–1445, 2019.

[230] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, "GraphIn: An Online High Performance Incremental Graph Processing Framework," in *Euro-Par 2016: Parallel Processing*, Springer, Springer International Publishing, 2016, pp. 319–333.

[231] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ACM, 2014, pp. 613–624.

[232] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: A system for real-time iterative analysis over evolving data," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 417–430.

[233] J. Shun, "Practical parallel hypergraph algorithms," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 232–249.

[234] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[235] J. Shun, L. Dhulipala, and G. Blelloch, "A simple and practical linear-work parallel algorithm for connectivity," in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, 2014, pp. 143–153.

[236] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Work-efficient parallel union-find with applications to incremental graph connectivity," in *European Conference on Parallel Processing*, Springer, 2016, pp. 561–573.

[237] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. Hsu, and K. Wang, "An overview of microsoft academic service (mas) and applications," in *Proceedings of the 24th International Conference on World Wide web*, 2015, pp. 243–246.

[238] G. M. Slota, J. W. Berry, S. D. Hammond, S. L. Olivier, C. A. Phillips, and S. Rajamanickam, "Scalable generation of graphs for benchmarking hpc community-detection algorithms," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[239] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010, pp. 939–948.

[240] G. L. Steele, D. Lea, and C. H. Flood, "Fast splittable pseudorandom number generators," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 453–472, Dec. 2014.

[241] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[242] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.

[243] B. Sun, T.-H. H. Chan, and M. Sozio, "Fully dynamic approximate k-core decomposition in hypergraphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 14, no. 4, pp. 1–21, 2020.

[244] T. Suzumura, S. Nishii, and M. Ganse, "Towards large-scale graph stream processing platform," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014, pp. 1321–1326.

[245] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International Scientific Conference and International Workshop Present Day Trends of Innovations*, vol. 1, 2012.

[246] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 49–66.

[247] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, Nov. 2013.

[248] L. Toader, A. Uta, A. Musaafir, and A. Iosup, "Graphless: Toward serverless graph processing," in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, IEEE, 2019, pp. 66–73.

[249] V. A. Traag, L. Waltman, and N. J. van Eck, "From Louvain to Leiden: Guaranteeing well-connected communities," *Scientific Reports*, vol. 9, no. 1, pp. 1–12, Mar. 2019.

[250] A. Uta, S. Au, A. Ilyushkin, and A. Iosup, "Elasticity in graph analytics? a benchmarking framework for elastic graph processing," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2018, pp. 381–391.

[251] M. P. Van Den Heuvel and O. Sporns, "Rich-club organization of the human connectome," *Journal of Neuroscience*, vol. 31, no. 44, pp. 15 775–15 786, 2011.

[252] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, IEEE, 2014, pp. 144–153.

[253] L. R. Varshney, B. L. Chen, E. Paniagua, D. H. Hall, and D. B. Chklovskii, "Structural properties of the caenorhabditis elegans neuronal network," *PLoS Comput Biol*, vol. 7, no. 2, e1001066, 2011.

[254] N. Veldt, A. R. Benson, and J. Kleinberg, "The Generalized Mean Densest Subgraph Problem," Aug. 2021.

[255] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in Facebook," in *Proceedings of the 2nd ACM Workshop on Online Social Networks*, ACM Press, 2009, pp. 37–42.

[256] K. Vora, R. Gupta, and G. Xu, "KickStarter, Fast and accurate computations on streaming graphs via trimmed approximations," in *Proceedings of the twenty-second International Conference on architectural support for programming languages and operating systems*, vol. 52, Association for Computing Machinery (ACM), May 2017, pp. 237–251.

[257] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[258] N. Wang, D. Yu, H. Jin, C. Qian, X. Xie, and Q.-S. Hua, "Parallel algorithm for core maintenance in dynamic graphs," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2017, pp. 2366–2371.

[259] T. Wang, *Integer hash function*, https://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm, 1997.

[260] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[261] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[262] C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Chelmis, and V. K. Prasanna, "Real-time analytics for fast evolving social graphs," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE, 2015, pp. 829–834.

[263] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A kvs for any scale," *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[264] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu, "Core decomposition in large temporal graphs," in *2015 IEEE International Conference on Big Data (Big Data)*, IEEE, 2015, pp. 649–658.

[265] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "Gram: Scaling graph computation to the trillions," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 408–421.

[266] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

[267] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[268] Y. Yang, D. Yan, H. Wu, J. Cheng, S. Zhou, and J. C. S. Lui, "Diversified Temporal Subgraph Pattern Mining," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1965–1974.

[269] M. Yu, D. Wen, L. Qin, Y. Zhang, W. Zhang, and X. Lin, "On querying historical k-cores," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2033–2045, 2021.

[270] W. W. Zachary, "An Information Flow Model for Conflict and Fission in Small Groups," *Journal of Anthropological Research*, vol. 33, no. 4, pp. 452–473, 1977.

[271] C. Zhang, F. Zhang, W. Zhang, B. Liu, Y. Zhang, L. Qin, and X. Lin, "Exploring Finer Granularity within the Cores: Efficient (k, p)-Core Computation," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, IEEE, 2020, pp. 181–192.

[272] F. Zhang, C. Li, Y. Zhang, L. Qin, and W. Zhang, "Finding critical users in social communities: The collapsed core and truss problems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 1, pp. 78–91, 2018.

[273] Y. Zhang and J. X. Yu, "Unboundedness and efficiency of truss maintenance in evolving graphs," in *Proceedings of the 2019 International Conference on Management of Data*, ACM, 2019, pp. 1024–1041.

[274] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, IEEE, 2017, pp. 337–348.

[275] W. Zhou, H. Huang, Q.-S. Hua, D. Yu, H. Jin, and X. Fu, "Core decomposition and maintenance in weighted graph," *World Wide Web*, pp. 1–21, 2020.