ABSTRACT

RANSHOUS, STEPHEN MICHAEL. Scalable Algorithms for Mining Dynamic Graphs and Hypergraphs with Applications to Anomaly Detection. (Under the direction of Dr. Nagiza F. Samatova.)

Graph data mining has become a ubiquitous tool for researchers and practitioners in numerous domains, including social sciences, financial markets, and computer security. In particular, mining dynamic graphs has gained substantial interest in the past decade, due to their robust expressiveness and natural ability to represent complex relationships that evolve over time. Traditional graph mining tasks, such as community detection, frequent subgraph mining, and anomaly detection, often operate on entire graph snapshots at a time, where a snapshot may consist of an entire day or week of activity. However, as the volume and velocity of data continues to increase, it is no longer feasible to analyze entire datasets at once, or even assume that the datasets will fit into memory. In this work we focus on developing efficient algorithms for mining dynamic graphs. We use anomaly detection as an exemplar task, but believe that our cross-cutting ideas and applications of efficient data structures apply much more broadly.

We propose two changes for how anomaly detection is performed over large-scale dynamic graphs to cope with the growing constraints. First, we transition from the traditional approach of analyzing graph streams, where each object in the stream is a full graph, to analyzing graph edge streams. An immediate benefit of this transition is a smaller unit of analysis, as well as more fine-grained attribution for anomalies. Second, instead of storing exact information, we utilize approximation, heuristics, and probabilistic data structures for retaining the salient information from the stream. This results in dramatically reduced time and space complexities for our algorithms. In this dissertation we develop anomaly detection algorithms for increasingly complex graph edge stream models and show their effectiveness, both theoretically and empirically.

In our first component, based on our extensive survey and gap analysis of the field, we begin with the simplest case, undirected graph edge streams. Key graph properties necessary for our anomaly detection algorithm are approximated from the stream using Count-Min (CM) sketches. Theoretical and empirical results show that the error of our approximations is minimal, while enabling constant time operations. Experiments on synthetic and real-world datasets show the effectiveness of our approach, yielding high precision and recall, as well as interesting case studies.

In our second component, we plan to extend the streaming model from graph edges to hypergraph edges, or hyperedges. As hyperedges represent higher order relationships, not strictly pairwise, we must transition to a more flexible notion of similarity and frequency, thus we can
no longer use CM sketches. Instead, we use MinHash and locality sensitive hashing to quickly and accurately estimate historical vertex co-occurrences. Experimental results show processing speedups between 33-750x compared to exact analysis, while maintaining comparable accuracy.

Finally, our last component examines the potential for pattern based anomalies in dynamic directed hypergraphs (dirhypergraphs). We perform a case study using the Bitcoin network, and propose an edge-based pattern which we posit may represent money laundering or transfer. Patterns in the dirhypergraph are found heuristically, and are centered around “exchange vertices”. Experimental results show that our laundering pattern appears at a significantly low rate, and that exchange vertices have several unique characteristics, enabling accurate classification models to be constructed.
DEDICATION

To my parents, for their unyielding support and encouragement. We made it.
BIOGRAPHY

Stephen Ranshous earned a Bachelor of Science in Computer Science and a minor in Electrical Engineering from the University of Florida in 2013. Upon graduation, he began his Doctor of Philosophy in Computer Science at North Carolina State University, with Dr. Nagiza Samatova as his advisor. During his doctoral studies, Stephen had research assistant appointments at Argonne National Laboratory and Pacific Northwest National Laboratory.
Completing this dissertation would not have been possible without the guidance, support, and encouragement from a number of people and institutions. Above all, I am eternally grateful to my advisor, Dr. Nagiza Samatova, without whom I would not have become the person and researcher I am today. Her enthusiasm and drive in has been infectious, and helped raise me to levels I did not think possible. By always placing her students first, she created a cooperative and collaborative environment that fostered intellectual growth, teamwork, and personal development.

In addition, I am grateful to all the faculty at NCSU that helped me along the way. In particular I would like to the members of my PhD committee, Professors Steffen Heber, Raju Vatsavai, and Dennis Bahler, for their valuable time and insightful feedback regarding my dissertation. I would also like to thank Dr. Douglas Reeves for my admission into the program and initial financial support.

Throughout my completion of this dissertation, I have been fortunate to have a number of collaborations with exceptionally gifted researchers, spread across top institutions across the nation. I would like to thank Tom Peterka at Argonne National Laboratory, for helping ease my transition from academics into my first true multi-disciplinary environment. I would like to thank Danai Koutra and Dr. Christos Faloutsos from CMU for their contributions to the survey performed in Chapter 2. From Pacific Northwest National Lab I would like to thank first and foremost Cliff Joslyn, for being a phenomenal mentor, host, and thought leader. Additionally, I would like to thank Sean Kreyling, Samuel Winters, Curt West, Svitlana Volkova, Sutanay Choudhury, Arun Sathanur, Katie Nowak, Brenda Praggastis, Chase Dowling, Amanda White, and Emilie Purvine. It is through this collaboration, and the computing resources provided by PNNL, that I was able to complete much of Chapter 5.

Additionally, I would like to thank the other members of Dr. Samatova’s research group for their support, comradery, and collaborations: Steve Harenberg, Mandar Chaudhary, Gonzallo Bello, Changsung Moon, Shiou “Claude” Tian Hsu, David “Drew” Boyuka. The work done in this dissertation, and many works outside the scope of this dissertation, would not have been possible without their help.

Finally, I would like to thank my family for their love and support over the years. Their constant encouragement has helped motivate me, and helped me push through challenging times.

This dissertation is based upon work supported in part by the Department of Energy National Nuclear Security Administration under Award Number(s) DE-NA0002576, the Laboratory for Analytic Sciences (LAS), the Scientific Discovery through Advanced Computing (SciDAC) program, the DOE SDAVI Institute, and the U.S. National Science Foundation (Expeditions
in Computing program). Any opinions or findings expressed in this material are those of the author(s) and do not necessarily reflect the views of any agent or entity of the US government.
**TABLE OF CONTENTS**

List of Tables ................................................................................. ix

List of Figures ................................................................................ xi

**Chapter 1 Introduction** ................................................................. 1

1.1 Graph Based Anomaly Detection .............................................. 3

1.1.1 Anomaly Detection in Dynamic Graphs ........................................ 4

1.1.2 Anomaly Detection in Edge Streams ............................................ 4

1.1.3 Anomaly Detection in Hyperedge Streams ................................. 5

1.1.4 Fraudulent Pattern Mining in Directed Hypergraphs ................. 6

1.2 Approximation, Heuristics, and Probabilistic Data Structures .......... 6

1.2.1 Approximation ............................................................................. 7

1.2.2 Heuristics ................................................................................ 7

1.2.3 Probabilistic Data Structures ...................................................... 7

**Chapter 2 Survey and Analysis of Anomaly Detection in Dynamic Graphs** . 8

2.1 Introduction .................................................................................. 8

2.2 Background ................................................................................... 9

2.3 Types of anomalies ...................................................................... 11

2.3.1 Type 1: Anomalous vertices ...................................................... 11

2.3.2 Type 2: Anomalous edges .......................................................... 12

2.3.3 Type 3: Anomalous subgraphs ................................................. 14

2.3.4 Type 4: Event and change detection ........................................... 15

2.4 Methods ....................................................................................... 17

2.4.1 Community Detection ................................................................. 18

2.4.2 Compression ............................................................................. 20

2.4.3 Matrix/Tensor Decomposition .................................................... 22

2.4.4 Distance Measures .................................................................... 24

2.4.5 Probabilistic Models ................................................................. 27

2.5 Discussion .................................................................................... 30

2.6 Conclusion ................................................................................... 33

**Chapter 3 Anomalies in Graph Edge Streams** .............................. 35

3.1 Introduction .................................................................................. 35

3.2 Problem Statement and Model ................................................... 36

3.2.1 Data Model ................................................................................ 37

3.2.2 Edge Scoring and Outlier Detection .......................................... 37

3.3 Model Approximation ................................................................. 40

3.3.1 The CountMin Sketch ............................................................... 41

3.3.2 Sample Score ........................................................................... 42

3.3.3 Preferential Attachment Score ................................................ 43

3.3.4 Homophily Score ................................................................. 44
3.3.5 Total Score .......................................................... 47

3.4 Complexity Analysis ................................................. 47
3.4.1 Time ................................................................. 48
3.4.2 Space ............................................................... 48

3.5 Experiments ........................................................... 48
3.5.1 Experimental Setup .............................................. 48
3.5.2 Dataset Descriptions ............................................ 49
3.5.3 Scalability .......................................................... 50
3.5.4 Identifying Injected Outliers ................................... 51
3.5.5 Real-World Outliers ............................................. 53

3.6 Related Work .......................................................... 55

3.7 Conclusion ............................................................. 55

Chapter 4 Anomalies in Hypergraph Edge Streams .................. 56
4.1 Introduction ........................................................... 56
4.2 Related Work ......................................................... 57
4.3 Problem Statement and Model ..................................... 58
4.3.1 Data Model ......................................................... 59
4.3.2 Hyperedge Scoring and Outlier Detection .................... 59
4.4 Model Approximation ................................................ 61
4.4.1 MinHash .......................................................... 61
4.4.2 LSH ................................................................. 63
4.4.3 Hyperedges Streams, MinHash, and LSH ..................... 64
4.5 Complexity Analysis ................................................. 65
4.5.1 Time ................................................................. 65
4.5.2 Space ............................................................... 66
4.6 Experiments ............................................................ 66
4.6.1 Dataset Descriptions ............................................ 66
4.6.2 Scalability .......................................................... 68
4.6.3 Identifying Injected Outliers ................................... 71
4.7 Conclusion ............................................................. 73

Chapter 5 Fraudulent Pattern Mining in the Bitcoin Directed Hypergraph .... 74
5.1 Introduction ........................................................... 74
5.2 Background ........................................................... 75
5.3 Bitcoin Transaction Motifs in a Directed Hypergraph ............ 76
5.4 Descriptive Statistics ................................................. 80
5.5 Classifying and Labeling Exchange Addresses .................... 84
5.6 Conclusions and Future Work ..................................... 88

Chapter 6 Conclusion ..................................................... 89
6.1 Future Work ........................................................... 90

BIBLIOGRAPHY ............................................................ 92
# APPENDICES

| Appendix A | Anomaly Detection Survey Summary Tables | 107 |
| Appendix B | Bitcoin Supplementary Info | 112 |
List of Tables

Table 2.1 Introductory and overview references ............................................. 10
Table 2.2 Notation Summary ........................................................................ 10

Table 3.1 Notation ......................................................................................... 41
Table 3.2 ROC AUC comparisons for vertex and edge weight sketch parameters
\(\varepsilon = 0.00001\), with the length of the shared sketch size shown above, all
sketches set \(\delta = 0.001\) ............................................................................. 51
Table 3.3 Approximation error as a result of using sketches. The values shown for
IMDB and DBLP are median \(\times 10^6 \pm \text{std} \times 10^3\), for the Enron datasets
(Full and Half datasets) the values shown are median \(\times 10^6 \pm \text{std} \times 10^9\). For
all runs, the vertex and edge weight sketches had \(\varepsilon = 10^{-6}\) and \(\delta = 10^{-3}\).
Median and standard deviation taken over 10 runs ........................................... 52

Table 4.1 Results of vertex independence tests for neighbors in each dataset. To test
if two vertices were independent of each other, we ran Fisher’s exact test
on their hyperedge incidence vectors and tested at a significance level of
0.05. Each dataset had 10,000 pairs of neighbors randomly sampled and
tested ............................................................................................................. 59
Table 4.2 Average run time and peak memory usage over 10 runs using the real-world
datasets. DNF indicates the method did not complete after 24 hours of run
time. The run time for the exact version is an order of magnitude or more
greater across all datasets, with DBLP not completing after 24 hours. Peak
memory usage shows that there is a tradeoff, drastically reducing run time
at the cost of memory. Within the approximation, using shorter MinHash
signatures clearly has a large impact on both run time and memory ............. 69
Table 4.3 ROC AUC for each method and dataset. Datasets are labeled E = Enron,
M = Movielens, I = IMDB, D = DBLP. Dataset suffix indicates injection
type: 1 percent outliers, 5 percent outliers, or Bursty outliers ....................... 72

Table 5.1 Participation of addresses in a 2-motif ............................................ 79
Table 5.2 Comparing labeled and unlabeled address’ degrees and weights ............. 82
Table 5.3 Results for exchange address classification. All results shown are mean +
\(-\text{std}\) over the 10 runs ............................................................................. 85
Table 5.4 Results for exchange vertex classification when features related to exchanges
are removed. All results shown are mean + \(-\text{std}\) over the 10 runs .............. 86
Table 5.5 Percent of addresses classified as exchanges .................................... 87

Table A.1 Algorithm Complexity Notation ................................................... 107
Table A.2 Summary of Methods. Checkmarks indicate type of anomaly found by each
method. *The complexity stated is, in some cases, a coarse approximation
provided for comparative purposes, and is for the entire graph series ......... 108
Table A.3 Summary of Methods. Checkmarks indicate type of anomaly found by each method. *The complexity stated is, in some cases, a coarse approximation provided for comparative purposes, and is for the entire graph series. . . . 109

Table A.4 Public Datasets; †Data may not already be in network format; *Synthetic instead of real data; ‡Many more datasets are available from the 2008, 2009, and 2010 challenges; ‡‡See [161] for one example of creating climate networks from raw data. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

Table A.5 Methods with open source code, the types of graphs they work on, the language they are written in, and the output format; *These methods operate on the community belongingness matrices of graphs, therefore they are applicable to any graph that can have communities extracted . . . . . 111
LIST OF FIGURES

Figure 1.1 Example graph containing two primary groups of co-authors, one group from computer science (blue), and one group from geriatric research (orange). Each author is a vertex, and edges connect authors who have co-published. The bridging edge between Stephen and Todd shows an interesting collaboration between computer science and geriatric research that may warrant further investigation. ................................. 2

Figure 1.2 Overview of the main dissertation Chapters, and how they incorporate our solutions to the driving research objectives of efficiency and increased model complexity. ................................. 3

Figure 1.3 Each figure shows an example of a different streaming model. Figure (a) shows a graph stream, where each incoming object is an entire graph snapshot, in this example representing a day of activity. Figure (b) shows an edge stream, where each incoming object is an individual graph edge, labeled with the time it was generated. Figure (c) shows a hyperedge stream, where each incoming object is a hyperedge, which may have any positive number of vertices incident on it. ................................. 5

Figure 2.1 Example of an anomalous vertex. A dynamic graph with two time steps showing a vertex, $v_6$, that is found to be anomalous based on its change in community involvement. Note that communities can be overlapping. In time step 1, $v_6$ is only part of community 2, whereas in time step 2 it is part of both communities 1 and 2. As no other vertices community involvement changes, $v_6$ is deemed anomalous. ................................. 12

Figure 2.2 An illustration of anomalous edges that occur because of an irregular pattern of their weight over time, with anomalous edges highlighted. At each time step, a vertex’s weight typically shifts by $\pm 0.05$ at most. However, edge (3, 4) has a spike in its weight at time step 2, unlike any other time in the series. Similarly, at time step 3, edge (1, 4) spikes in value. These spikes cause the edges to be considered anomalous ................................. 13

Figure 2.3 Two different types of anomalous subgraphs. Figure 2.3a shows a shrunken community, where a community loses members from one time step to the next. Figure 2.3b shows a split community, where a single community breaks into several distinct smaller communities. ................................. 14

Figure 2.4 Illustration of the difference between an event and change point. An event (Figure 2.4a) is isolated in time, unlike the preceding or subsequent time point. A change point (Figure 2.4b) marks a point in time where the graph undergoes a substantial change, but the change then persists. ................................. 16

Figure 3.1 Example graph and table illustrating the scores each edge incident on vertex 2 would receive if it were to arrive next in the stream. For calculations, $w(0, 2) = 3$, all other edge weights are 1, $\alpha + \beta = 0.75$, and $\gamma = 0.25$. Note how the homophily score complements the evidence-based scores. ................................. 39
Figure 3.2 How we use a CM sketch for approximating edge weights. ...................................... 42
Figure 3.3 Run times for various approximation parameters compared to the exact implementation. For all approximations, the vertex and edge weight sketches had $\varepsilon = 10^{-6}$ and $\delta = 10^{-3}$. The shared sketch heuristic runs both had a length of $l = \lceil \frac{e}{10000} \rceil$. ................................................. 50

Figure 4.1 Example of scoring hyperedges in a stream using hard matching. Hard matching is too constrained for obtaining meaningful results. Even though vertices $\{1, 3\}$ co-occur in $he_1$, $he_2$ is given an outlier score of 1. Similarly, $he_3$ is given a score of 1 even though $\{2, 3\}$ co-occur in $he_1$. .............................................................. 60

Figure 4.2 Example of scoring hyperedges in a stream using soft matching with $\tau = 0.5$. The co-occurrences of vertices is now taken into account, and the hyperedges $he_2$ and $he_3$ are deemed normal. .............................................. 61

Figure 4.3 How many times each hyperedge is repeated, i.e. it’s count, in the Enron dataset. Figure 4.3a shows hard count, Figure 4.3c shows soft count with $\tau = 0.25$. ................................................................. 62

Figure 4.4 Example of computing a single MinHash value for a set. The MinHash value is the minimum value of the set with respect to the chosen hash function $h$. .......................................................... 63

Figure 4.5 MinHash signatures are formed by $k$ minhash values, each generated by a different hash function. Signatures are then broken into $b$ bands of length $r$ for storing in the hash tables comprising LSH index. In this figure we show a single hash table for every band, but in reality each band has its own hash table. ................................................................. 64

Figure 4.6 Average time taken to process “batches” of 1,000 hyperedges. No model uses batch analysis, times were simply aggregated into batches for illustrative purposes. In DBLP the Exact versions were unable to complete in 24 hours, processing less than half the stream. ................................................................. 68

Figure 4.7 Memory used by each model over time. The exact models use the same amount of memory, so one time series line is not visible. “Batches” are used for illustrative purposes only, each batch representing the processing of 1,000 hyperedges. ................................................................. 70

Figure 5.1 (Left) Bitcoin transactions as a labeled multigraph. (Right) Bitcoin transactions as a bipartite multigraph. ................................................................. 76

Figure 5.2 Bitcoin transactions as a directed hypergraph. ................................................................. 77

Figure 5.3 Graph motifs: (Far Left) The single undirected graph 2-motif (above) with its three directed motifs $\alpha, \beta$, and $\gamma$ (below), including the two isomorphic $\alpha, \alpha'$ patterns. (Right) The three undirected 3-motifs and their directed versions. ................................................................. 77

Figure 5.4 A 2-motif in a dirhypergraph: Two transactions $E_1, E_2$ with sets of tail and head vertices $T_1, H_1, T_2, H_2$ respectively. (Left) Generic. (Center) A linear STB. (Right) A circular STB. ................................................................. 78

Figure 5.5 (Left) Generic 2-motif. (Right) Instantiated with counts for days 2200-2299 of our data. For details about our dataset, see Section 5.4. ................................................................. 79
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Daily activity in each dirhypergraph.</td>
<td>81</td>
</tr>
<tr>
<td>5.7</td>
<td>Cumulative percent of addresses with total (left) or average (right) input and output weights.</td>
<td>83</td>
</tr>
<tr>
<td>5.8</td>
<td>Counts (left) and proportions (right) of labeled addresses on the inputs and outputs of STBs.</td>
<td>83</td>
</tr>
<tr>
<td>5.9</td>
<td>Distribution of how many exchange addresses siblings (left) or total siblings (right) each address has. These distributions were drawn from a random sample of 100K exchanges and 100K non-exchanges.</td>
<td>85</td>
</tr>
<tr>
<td>B.1</td>
<td>Frequency of exchanges in STBs.</td>
<td>113</td>
</tr>
<tr>
<td>B.2</td>
<td>Relationship between the number of vertices owned by each exchange and the frequency of that exchange in STBs.</td>
<td>114</td>
</tr>
</tbody>
</table>
Graphs have gained immense popularity in the research and data science communities, given their ability to represent complex systems and relationships. Each object in the system is a vertex in the graph, and relationships among the objects form the edges of the graph, connecting vertices. For example, in climate science, regions of the Earth are represented as vertices, and edges connect vertices (regions) which have highly correlated weather variables, such as sea surface temperature. Here edges have a physical interpretation—the correlation between regions of the Earth—but this is not always the case. In social networks, where users are vertices in the graph, edges can represent more abstract relationships, such as friendship, communication, or shared interests. Figure 1.1 shows an example of a co-author graph, where vertices are researchers who have published papers, and two authors share an edge if they have co-published before.

Graph mining techniques can provide novel insight into the graphs, and hence the represented systems. In the past decade focus has shifted from static graphs to dynamic graphs, which allow changes to their structure and attributes over time. However, primarily as a result of technology growing cheaper, more available, and pervasive, researchers and analysts are becoming inundated with data and facing new challenges when trying to mine large dynamic graphs. For example, in 2013 Twitter was receiving 500 million new tweets per day [98], each of which introduces potentially several new edges in the network. The vast majority of algorithms for tasks or queries such as community detection [60, 151], frequent subgraph mining [27, 177], and anomaly detection [9, 140], operate on sequences of graph snapshots. As graph snapshots can comprise an
entire day, week, or year of activity, many assumptions, such as the graph fitting into memory, or that $O(n^2)$ algorithms will terminate in reasonable time, must be revisited. In this dissertation we focus on developing efficient ways of analyzing large-scale dynamic graphs, with a focus on techniques for anomaly detection. While our specific focus is anomaly detection, we believe the approaches proposed for retaining salient information about the underlying graph is applicable to many other tasks.

Anomaly detection, also referred to as outlier detection, tries to identify data, here graph structures, that are rare, isolated, and/or surprising [11]. For example, edges that act as a “bridge” between groups of vertices often indicate interesting phenomena [5, 58, 139], such as multi-disciplinary collaborations among researchers from otherwise disparate fields, as shown in Figure 1.1. A similar idea has been leveraged to identify unlikely or disallowed paths in system call and program execution graphs, for the purpose of intrusion detection [184]. We focus on developing algorithms for accurate and efficient anomaly detection that directly address two challenges in particular: (1) the increased complexity of the systems being modeled; (2) the growing gap between computing and storage capability, and the volume of data being generated. We address these challenges by designing our algorithms around two cross-cutting ideas. First, we transition from analyzing graph streams and series, to edge streams. Second, recognizing that exact analysis is often no longer feasible, we leverage approximation, heuristics, and probabilistic data structures. Specifically, in this dissertation we explore solutions for the following three research objectives.

1. Efficient anomaly detection in edge streams (Chapter 3)

2. Efficient anomaly detection in hyperedge streams (Chapter 4)
Figure 1.2 Overview of the main dissertation Chapters, and how they incorporate our solutions to the driving research objectives of efficiency and increased model complexity.

3. Fraudulent pattern mining in the dynamic Bitcoin directed hypergraph (Chapter 5)

Figure 1.2 illustrates how each research objective addresses the two cross-cutting challenges outlined above. In Chapter 2 we perform an extensive survey of anomaly detection in dynamic graphs. From this, we identify two important challenges, which lay the foundation for this dissertation. Chapter 3 proposes the first algorithm for outlier detection in graph edge streams. In addition to a fundamentally sound exact model based on link prediction techniques, we show an efficient approximation algorithm that uses probabilistic data structures to provide theoretical error bounds. Similar to Chapter 3, Chapter 4 is the first proposed algorithm for identifying outliers in hyperedge streams. The increased complexity of transitioning from edges to hyperedges necessitates a new form of approximation be used. Finally, Chapter 5 performs a case study on fraudulent pattern mining in the Bitcoin directed hypergraph. Again we increase the complexity, from undirected to directed hypergraphs, but this time rely on heuristics instead of approximation for identifying patterns of interest.

1.1 Graph Based Anomaly Detection

Given the decades of research into techniques for identifying anomalies, outliers, or abnormalities, it is natural to wonder why graph based methods are necessary. Perhaps the single largest reason is leveraging how graphs represent the data, compared to traditional multi-dimensional point clouds. Inter-dependencies and relationships are explicitly modeled and considered, instead of treating each data point independently. Graphs naturally capture long-range correlations, non-linear relationships, and even high order (more than pairwise) relationships.
With these advantages, however, come additional challenges. Given such a rich model, it is perhaps obvious that anomalies may be defined in a nearly endless number of ways. Anomalies could be defined at the vertex, edge, or even subgraph level, using topological, attribute, or label information, and be defined locally, e.g. comparing a vertex to its neighbors, or globally, e.g. comparing a vertex to every other vertex in the graph. Moreover, even in plain graphs, which have no attributes or label information on vertices and edges, enumerating the search space is a combinatorially challenging problem. As the substructure(s) of interest grow in complexity, the search space grows superlinearly, in the worst case exponentially.

1.1.1 Anomaly Detection in Dynamic Graphs

In the past decade, focus has shifted from static to dynamic graphs. Unlike static graphs, dynamic graphs allow for modifications to their structure and attributes. For example, in a dynamic graph edges can be inserted or deleted, where a static graph is a snapshot of the system in time, therefore the edge set is fixed. When considering the dynamic nature of the data, new challenges are introduced, such as new types of anomalies, increased storage and processing requirements, and differentiating between organic graph evolution and slow-to-develop anomalies. In Chapter 2 we perform a qualitative comparison and gap analysis of the existing methods. Chapter 2 lays the foundation for the rest of our dissertation, by defining classes of anomalies in dynamic graphs, identifying and mathematically formalizing a two-stage approach that is common among all papers we review, and most importantly identifying open questions and areas of research. The findings of this Chapter were published in [140].

1.1.2 Anomaly Detection in Edge Streams

All the methods examined in Chapter 2 represented a dynamic graph using streams or series of graph objects, i.e. each object in the stream was a (static) graph snapshot. Analyzing entire graph snapshots at a time, and comparing adjacent snapshots in the stream, has several drawbacks. First, it requires that entire graphs be stored in main memory for analysis. Second, much or all of the historical information from the stream is lost, as only the most recent snapshots are considered. Third, snapshot comparison often precludes, or severely hinders, the ability to perform fine-grained attribution of detected anomalies. In response to these limitations, we propose a transition from graph stream analysis, to edge stream analysis, i.e. each object in the stream is a graph edge. Figure 1.3 illustrates the differences between the streaming models discussed.

In Chapter 3 we explore a method for identifying anomalies in edge streams. As edge stream mining is a more fine-grained analysis of the changes, we hypothesize it is more much amenable to attribution. However, addressing memory constraints and historical information loss issues is
still a challenging problem. We propose a model for calculating an outlier score for each edge in the stream, and utilize Count-Min (CM) sketches to approximate the structural properties of the stream that are relevant to our model. Using CM sketches, we are able to prove theoretical probabilistic error bounds on our edge scores. Results on synthetic and real-world datasets demonstrate empirically the low error rate resulting from approximation, and ability of our method to identify outliers—both quantitatively and qualitatively. All findings from this chapter were published in [139].

1.1.3 Anomaly Detection in Hyperedge Streams

Restricting, or reducing, edges to be pairwise does not faithfully represent numerous types of relationships and interactions, and results in a loss of information. For example, if three researchers co-author a paper together, a graph represents this with three pairwise edges. In Chapter 3 this would result in three new edges in the stream. However, it is possible this same edge configuration may have been the result of every author pair publishing a paper together, and never having published a paper with all three of them. Hyperedges, which can be incident on any positive number of vertices, enable higher order relationships such as these to be captured. In Chapter 4 we propose a method for identifying outliers in hyperedge streams.

In addition to the challenges enumerated in Chapter 3 for edge streams, hyperedge streams face an even larger combinatorial issue as the number of unique hyperedges possible in a graph with \( n \) vertices is \( 2^n \). Again we turn to approximation and efficient data structures to
combat these challenges. In particular, we use MinHash and locality sensitive hashing (LSH) to facilitate efficient storage and analysis of the stream. Experimental results show that our model is effective in identifying outliers, and support our hypothesis that exact hyperedge matching is too constrained. Moreover, our approximation provides stream processing speedups between 33-750x compared to exact analysis, while maintaining comparable accuracy. The results of this work were published in [141].

1.1.4 Fraudulent Pattern Mining in Directed Hypergraphs

As edges can be directed and undirected in graphs, hyperedges can be directed and undirected in hypergraphs. Directed edges, or arcs, have a single tail vertex and a single head vertex. Directed hyperedges, which we will call hyperarcs from here onward, have a set of tail vertices and a set of head vertices. Directed hypergraphs have been used to model a wide array of domains, from propositional logic [61] to chemical reactions [68]. In Chapter 5 we perform a case study in the Bitcoin network for identifying and reporting potential fraudulent behavior.

Bitcoin is most naturally represented as a directed hypergraph, with each transaction generating a hyperarc in the network. In collaboration with financial forensic experts, we consider what money laundering behavior may look like in the directed hypergraph model. Our analysis focuses on two central topics: exchanges, the entrance and exit from the Bitcoin world; and a specific 2-path pattern we posit could be laundering behavior. Results show we are capable of identifying exchanges in the network with over 80% accuracy using purely structural features of the graph, and that our chosen 2-pattern is exceptionally rare compared to the number of 2-paths in the dirhypergraph. More details can be found in the original publication [142].

1.2 Approximation, Heuristics, and Probabilistic Data Structures

When designing algorithms for graph mining, it is important to always bear in mind the real-world limitations imposed by computing capability, memory availability, and the enormity of data. Ideally, we would like to have an exact model of the graph, or the properties we need from it, and perform our analytics on this model, yielding accurate solutions. However, for many problems or datasets, this is not feasible. Given a finite amount of resources, as the amount of data being analyzed increases, the complexity of the analytics must commensurately decrease. For example, on a graph with only 10 vertices we may be able to solve the traveling salesman problem, but on a graph with 1 billion vertices we may only be able to query for the degree distribution. But what if we want to answer challenging questions about large scale datasets? Techniques such as approximation, heuristics, and probabilistic data structures must
be leveraged to trade accuracy, precision, or completeness of the solution, with the time and space required to find a solution.

1.2.1 Approximation

We differ in how we use approximation from the conventional use in computer science. Traditionally, approximation refers to algorithms which solve optimization problems, most commonly NP-hard problems, within a provable error bound. A canonical example is a 2-approximation for the minimum vertex cover of a graph, meaning the solution found is guaranteed to be at most twice as large as an optimal solution. On the other hand, when we talk about approximation, we will use it in the context of approximating graph properties or count values. For example, if we want to approximate the degree of a vertex we will try to constrain it to a fixed, known bound. If $deg(u)$ is the true degree of a vertex, we will bound the approximation $deg_e(u)$ and use the following notation $deg(u) \leq deg_e(u) \leq deg(u) + \epsilon$.

1.2.2 Heuristics

Heuristics are decision making rules chosen for speed, practicality, and intuition, over the global optimality of the choice. While approximation algorithms may employ heuristics, heuristics in general have no requirement to provide error bounds. One of the main heuristics we employ is setting thresholds for labeling outliers. More sophisticated statistical methods, such as autoregressive moving averages, could be used for calculating outlier threshold values; however, we opt to simply set a threshold value, e.g. $score < 0.05 \rightarrow$ outlier.

1.2.3 Probabilistic Data Structures

Deterministic data structures, such as hash tables, quickly become too cumbersome when analyzing large or streaming datasets. While they are excellent tools for answering queries such as unique item count or frequency count, memory quickly becomes a limiting factor. Instead, we can use probabilistic data structures, which, similar to approximation algorithms, provide a way of trading accuracy for memory. The more memory used, the more accurate your model or query result. Consider estimating the frequency of an item in a stream. A straightforward approach would be to keep a mapping from item to count, and each time a new item is processed in the stream you increment the count associated with it. The size of the map is linear in the number of unique items, which both may not be known a priori, and may scale beyond the capacity of available memory. Instead, we could create a fixed size array of counters, and hash each item label to a cell in the array. The memory required is now constant, at the cost of potentially having hash collisions. Item frequencies are now returned as an estimated value, and the probability that the estimated value is within some degree of accuracy.
SURVEY AND ANALYSIS OF ANOMALY DETECTION IN DYNAMIC GRAPHS

2.1 Introduction

Anomaly detection is an important problem with multiple applications, and thus has been studied for decades in various research domains. In the past decade there has been a growing interest in anomaly detection in data represented as graphs, largely due to their robust expressiveness and their natural ability to represent complex relationships. Examples include global financial systems connecting banks across the world, electric power grids connecting geographically distributed areas, and social networks that connect users, businesses, or customers using relationships such as friendship, collaboration, or transactional interactions. These are examples of dynamic graphs, which, unlike static graphs, are constantly undergoing changes to their structure or attributes. Possible changes include insertion and deletion of vertices (objects), insertion and deletion of edges (relationships), and modification of attributes (e.g., vertex or edge labels).

An important problem over dynamic graphs is anomaly detection – finding objects, relationships, or points in time that are unlike the rest. There are many high-impact and practical applications of anomaly detection spanning numerous domains. A small sample includes: detection of ecological disturbances, such as wildfires [43, 44] and cyclones [42]; intrusion detection for individual systems [184] and network systems [51, 66, 182]; identifying abnormal users and
The ubiquitousness and importance of anomaly detection in dynamic graphs has led to the emergence of dozens of methods over the last five years (see Tables A.2 and A.3). These methods complement techniques for static graphs [12, 54, 55, 125], as the latter often cannot be easily adopted for dynamic graphs. When considering the dynamic nature of the data, new challenges are introduced, including:

- New types of anomalies are introduced as a result of the graph evolving over time, for example, splitting, disappearing, or flickering communities.

- New graphs or updates that arrive over time need to be stored and analyzed. Storing all the new graphs in their entirety can vastly increase the size of the data. Therefore, typical offline analysis, where multiple passes over the data are acceptable and all of the data are assumed to fit into memory, becomes infeasible. Conversely, storing only the most recent graph or updates restricts the analysis to a single point in time.

- Graphs from different domains, such as social networks compared to gene networks, may exhibit entirely different behavior over time. This divergence in evolution can lead to application-specific anomalies and approaches.

- Anomalies, particularly those that are slow to develop and span multiple time steps, can be hard to differentiate from organic graph evolution.

Although anomaly detection has been surveyed in a variety of domains [23, 34–36, 80], it has only recently been touched upon in the context of dynamic graphs [9, 26, 77]. In this chapter, we hope to bridge the gap between the increasing number of methods for anomaly detection in dynamic graphs and the lack of their comprehensive analysis. In particular, we formalize four types of anomalies in dynamic graphs, develop and fit a taxonomy to all methods analyzed, identify a two-step approach commonly utilized, and qualitatively compare dozens of state-of-the-art methods.

### 2.2 Background

Anomaly, or outlier, detection is a problem that spans many domains. Chandola et al. [34] provide an excellent overview, taxonomy, and analysis of a multitude of techniques (e.g., classification, clustering, statistical) for a variety of domains, expanding the work of Hodge et al. [84] and Agyemang et al. [6]. As our focus will be on graphs, it is important to have a basic understanding of graph theory. West et al. [181] and Balakrishnan et al. [20] both offer comprehensive and approachable introductions to graph theory, covering all of the basics required for this chapter.
and well beyond. Cook and Holder [47] show how the theoretical concepts can be applied for graph mining, and Samatova et al. [155] provide an overview of many graph mining techniques as well as implementation details in the R programming language\footnote{Information on the R programming language can be found at http://www.r-project.org/}. For brevity we do not provide an introduction to the fundamentals of the types of methods we discuss, so we provide references for introductory and overview material for each of them in Table 2.1.

It is important to note that in many domains the data are naturally represented as a graph, with the vertices and edges clearly defined. However, in some cases, how to represent the data as a graph is unclear and can depend on the specific research question being asked. The conversion processes used in specific domains are outside the scope of this work, and we assume all data are already represented as dynamic graphs.

Table 2.2 Notation Summary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>Graph series with a fixed number of time points</td>
</tr>
<tr>
<td>$G_t$</td>
<td>Snapshot of the graph series at time $t$</td>
</tr>
<tr>
<td>$G^s$</td>
<td>The $s^{th}$ graph segment, a grouping of temporally adjacent graphs</td>
</tr>
<tr>
<td>$I_t$</td>
<td>Vertex partitioning at time $t$, separating all the vertices into disjoint sets</td>
</tr>
<tr>
<td>$V_t$</td>
<td>Vertex set for the graph at time point $t$</td>
</tr>
<tr>
<td>$v_i$</td>
<td>Vertex $i$</td>
</tr>
<tr>
<td>$E_t$</td>
<td>Edge set for the graph at time point $t$</td>
</tr>
<tr>
<td>$e_{i,j}$</td>
<td>Edge between $v_i$ and $v_j$</td>
</tr>
<tr>
<td>$c_0$</td>
<td>Threshold value for normal versus anomalous data</td>
</tr>
</tbody>
</table>
2.3 Types of anomalies

In this section, we identify and formalize four types of anomalies that arise in dynamic graphs. These categories represent the output of the methods, not the implementation details of how they detect the anomalies, e.g., comparing consecutive time points, using a sliding window technique. Note that often times in real-world graphs (e.g., social and biological networks) the vertex set $V$ is called a set of nodes. However, to avoid confusion with nodes in a physical computer network, and to align with the abstract mathematical representation, we call it a set of vertices.

Because the graphs are assumed to be dynamic, vertices and edges can be inserted or removed at every time step. For the sake of simplicity, we assume that the vertex correspondence and the edge correspondence across different time steps is resolved due to unique labeling of vertices and edges, respectively. We define a graph series $G$ as an ordered set of graphs with a fixed number of time steps. Formally, $G = \{G_t\}_{t=1}^T$, where $T$ is the total number of time steps, $G_t = (V_t, E_t \subseteq (V_t \times V_t))$, and the vertex set $V_t$ and edge set $E_t$ may be plain or attributed (labeled). Graph series where $T \to \infty$ are called graph streams. In the following subsections we start with an intuitive explanation of the problem definition, then give a general formal definition of the anomaly type, continue with some applications, and conclude with a representative example.

2.3.1 Type 1: Anomalous vertices

Anomalous vertex detection aims to find a subset of the vertices such that every vertex in the subset has an “irregular” evolution compared to the other vertices in the graph. Optionally, the time point(s) where the vertices are determined to be anomalous can be identified. What constitutes irregular behavior is dependent on the specific method, but it can be generalized by assuming that each method provides a function that scores each vertex’s behavior, e.g., measuring the change in the degree of a vertex from time step to time step. In static graphs, the single snapshot allows only intra-graph comparisons to be made, such as finding vertices with an abnormal egonet density [12]. Dynamic graphs allow the temporal dynamics of the vertex to be included, introducing new types of anomalies that are not present in static graphs. A high level definition for a set of anomalous vertices is as follows.

**Definition 2.1** (Anomalous vertices). Given $G$, the total vertex set $V = \bigcup_{t=1}^T V_t$, and a specified scoring function $f : V \to \mathbb{R}$, the set of anomalous vertices $V' \subseteq V$ is a vertex set such that $\forall v' \in V', |f(v') - \hat{f}| > c_0$, where $\hat{f}$ is a summary statistic of the scores $f(v)$, $\forall v \in V$.

An example of an anomalous vertex is shown in Figure 2.1. Two time steps are shown, both with 10 vertices and two communities. In this example, anomalous vertices are those that have
Figure 2.1 Example of an anomalous vertex. A dynamic graph with two time steps showing a vertex, $v_6$, that is found to be anomalous based on its change in community involvement. Note that communities can be overlapping. In time step 1, $v_6$ is only part of community 2, whereas in time step 2 it is part of both communities 1 and 2. As no other vertex’s community involvement changes, $v_6$ is deemed anomalous.

a substantial change in their community involvement compared to the rest of the vertices in the graph. As $v_6$ is the only vertex that has any change in its community involvement, it is labeled as an anomaly. Formally, we can measure the change in community involvement between adjacent time steps $t$ and $t-1$ by letting $f(v) = \sum_{i=1}^{\left|C\right|} c_{t,i,v}^{t-1} \oplus c_{t,i,v}^{t}$, where $C = \{ c_1, c_2, \ldots, c_{\left|C\right|} \}$ is the set of communities, $c_{t,i,v}^{t} = 1$ if $v$ is part of community $c_i$ at time step $t$ and 0 otherwise, and $\oplus$ is the xor operator.

The scoring function $f$ will depend on the application. In the example shown in Figure 2.1, the vertices were scored based on the change in community involvement. However, if the objective is identifying computers on a network that become infected and part of a botnet, then an appropriate scoring function might be measuring the change in the number of edges each vertex has between time steps, or the change in the weights of the edges.

Typical applications of this type of anomaly detection are identifying vertices that contribute the most to a discovered event (also known as attribution), such as in communication networks [7], and observing the shifts in community involvement [63, 88].

2.3.2 Type 2: Anomalous edges

Similar to vertex detection, edge detection aims to find a subset of the edges such that every edge in the subset has an “irregular” evolution, optionally identifying the time point(s) where they are abnormal. Again, this concept can be generalized by assuming each method employs a function that maps each edge in the graph to a real number, low values indicating unusual behavior. In a static graph, a distribution of the edge weights can be found, and each edge can be assigned a score according to the probability of its weight. However, due to the temporal nature of dynamic graphs, two new main types of irregular edge evolution can be found: (i)
abnormal edge weight evolution [105], where the weight of a single edge fluctuates over time and has inconsistent spikes in value, and (ii) appearance of unlikely edges in a graph between two vertices that are not typically connected or part of the same community [1, 143]. Anomalous edge detection can be formally defined as follows.

**Definition 2.2** (Anomalous edges). Given \( G \), the total edge set \( E = \bigcup_{t=1}^{T} E_t \), and a specified scoring function \( f : E \rightarrow \mathbb{R} \), the set of anomalous edges \( E' \subseteq E \) is an edge set such that \( \forall e' \in E' \), \( |f(e') - \hat{f}| > c_0 \), where \( \hat{f} \) is a summary statistic of the scores \( f(e), \forall e \in E \).

Figure 2.2 An illustration of anomalous edges that occur because of an irregular pattern of their weight over time, with anomalous edges highlighted. At each time step, a vertex’s weight typically shifts by \( \pm 0.05 \) at most. However, edge (3, 4) has a spike in its weight at time step 2, unlike any other time in the series. Similarly, at time step 3, edge (1, 4) spikes in value. These spikes cause the edges to be considered anomalous.

Figure 2.2 shows an example of edges that exhibit irregular edge weight evolutions. Over the five time steps there are two anomalous edges, both having a single time point that is unlike the rest of the series. Letting \( f(e) = |w_t(e) - w_{t-1}(e)| + |w_t(e) - w_{t+1}(e)| \), where \( w_t(e) \) is the edge weight at time step \( t \), each edge is scored based on its current weight compared to the previous and following weight. Abnormally large changes in the weight of an edge results in it being flagged as anomalous. For example, at time step 2, using the mean change in weights \( \hat{f} = 0.43 \) as a summary statistic and \( c_0 = 0.10 \) as a threshold, results in edge (3, 4) being declared anomalous.

While our example used a simple thresholding approach for edge weights, various other scoring functions can be used. Finding unlikely social interactions can be accomplished by modeling the edges between people as the number of times they communicate daily, and then scoring each edge at each time step based on the probability of it (1) being present, and (2) having a particular weight [82]. Another application is identifying anomalous traffic patterns in a graph where vertices represent road intersections and edges represent the roads themselves. Projecting the edges into a feature space based on their average speed of traffic throughout the day, the pairwise similarity between every edge can be calculated. Each edge is then scored.
based on the change in the similarity between itself and every other edge [105].

### 2.3.3 Type 3: Anomalous subgraphs

Finding subgraphs with irregular behavior requires an approach different from the ones for anomalous vertices or edges, since enumerating every possible subgraph in even a single graph is an intractable problem. Hence, the subgraphs that are tracked or identified are typically constrained, for example, to those found by community detection methods. In these cases, matching algorithms are required to track the subgraphs across time steps, such as the community matching technique used in [70]. Once a set of subgraphs has been determined, intra-graph comparisons or inter-time point comparisons can be made to assign scores to each subgraph, e.g., measuring the change in the total weight of the subgraph between adjacent time steps. Typical intra-graph comparison methods, such as finding unique substructures in the graph that inhibit compressibility [125], must be extended to incorporate the additional information gained by using a dynamic network. Instead of finding structures that are unique within a single graph, structures that are unique to a single graph within a series of graphs can be found. Anomalies of this type, unique to dynamic networks, include communities that split, merge, disappear and reappear frequently, or exhibit a number of other behaviors.

**Definition 2.3 (Anomalous subgraphs).** Given \( G \), a subgraph set \( H = \bigcup_{t=1}^{T} H_t \) where \( H_t \subseteq G_t \), and a specified scoring function \( f : H \rightarrow \mathbb{R} \), the set of anomalous subgraphs \( H' \subseteq H \) is a subgraph set such that \( \forall h' \in H', |f(h') - \hat{f}| > c_0 \), where \( \hat{f} \) is a summary statistic of the scores \( f(h) \), \( \forall h \in H \).

![Figure 2.3](image)

**Figure 2.3** Two different types of anomalous subgraphs. Figure 2.3a shows a shrunken community, where a community loses members from one time step to the next. Figure 2.3b shows a split community, where a single community breaks into several distinct smaller communities.

Two specific types of anomalous subgraphs are shown in Figure 2.3. A shrunken community,
Figure 2.3a, is when a single community loses a significant number of its vertices between time steps. Assuming that a matching of communities between adjacent time steps $t$ and $t-1$ is known, shrunken communities can be identified by measuring the decrease in the number of vertices in the community, $f(h) = |h_t \cap h_{t-1}| - |h_{t-1}|$. Figure 2.3b is an example of a split community, when a single community divides into several distinct communities. Split communities will have a high matching score or probability for more than one community in the next graph of the series. Therefore, split communities can be found by examining two adjacent time steps and letting $f$ score each community in $t-1$ as the number of communities it is matched to in $t$.

What constitutes an anomalous subgraph is heavily dependent on the application domain. Automatic identification of accidents in a traffic network can be done by letting edge weights represent outlier scores, then mining the heaviest dynamic subgraph [119]. Similarly, changes and threats in social networks can be found by running community detection on a reduced graph composed of suspected anomalous vertices [82] and performing an eigen decomposition on a residual graph [116].

### 2.3.4 Type 4: Event and change detection

Unlike the previous three types of anomalies, the two types discussed here are exclusively found in dynamic graphs. We start by defining the problem of event detection, which has attracted much interest in the data mining community. Event detection has a much broader scope compared to the previous three types of anomalies, aiming to identify points in time that are significantly different from the rest. Isolated points in time where the graph is unlike the graphs at the previous and following time points represent events. One approach to measuring the similarity of two graphs is comparing the signature vector of summary values extracted from each graph, such as average clustering coefficient. The task of attribution—finding the cause for the event—is not required as part of event detection, and is often omitted. However, once the time points for events have been identified, potential causes can be found by applying techniques for anomaly detection in static graphs [12, 54, 55, 125].

**Definition 2.4** (Event detection). Given $G$ and a scoring function $f: G_t \rightarrow \mathbb{R}$, an event is defined as a time point $t$, such that $|f(G_t) - f(G_{t-1})| > c_0$ and $|f(G_t) - f(G_{t+1})| > c_0$.

One of the simplest similarity metrics for graphs is comparing the number of vertices and edges in them. In Figure 2.4a, the event at time step 3 can be found by counting the edges in each graph, $f(G_t) = |E_t|$, and comparing the adjacent time points. The number of edges in each graph, \{6, 6, 14, 7, 7\}, shows that there is a significant change in the structure of the graph at time step 3, almost forming a clique. Note that this time point is isolated, with the surrounding time points being very dissimilar, indicating an event occurred.
Event detection, while providing less specific information than vertex, edge, or subgraph detection, is highly applicable in many areas. Approximating the data available using a tensor decomposition, then scoring the time point as the amount of error in the reconstruction, has been shown to effectively detect moments in time when the collective motions in molecular dynamics simulations change [138]. Other applications include finding frames in a video that are unlike the others [154, 174], and detecting disturbances in the ecosystem (e.g., wildfires) [43].

Now, we move on to the problem of change detection, which is complementary to event detection. It is important to note the distinction between event and change detection. While events represent isolated incidents, change points mark a point in time where the entire behavior of the graph changes, and the difference is maintained until the next change point. Figure 2.4 illustrates the difference between the two. The distinction between events and change points manifests in the modification of the constraint that the value of the scoring function at \( t \) be more than a threshold value away from both \( t - 1 \) and \( t + 1 \), to being more than a threshold value away from only \( t - 1 \).

**Definition 2.5** (Change detection). Given \( G \) and a scoring function \( f : G_t \rightarrow \mathbb{R} \), a change is defined as a time point \( t \), such that \( |f(G_t) - f(G_{t-1})| > c_0 \) and \( |f(G_t) - f(G_{t+1})| \leq c_0 \).

An example of change detection is shown in Figure 2.4b. Similar to event detection, this change can be identified by comparing the number of edges at adjacent time points. Just as in Figure 2.4a, a number of edges are added to the graph, forming nearly a clique. However,
at time step 4, instead of the graph reverting back to its original structure, the added edges are persistent and the graph has a new “normal” behavior. The persistence of the new edges indicates that at \( t = 3 \) a change was detected, not an event.

One of the most popular applications of change detection is in networks modeling human interactions, such as communication and co-authorship networks. Authors that act as a “bridge” between different conferences, or switch areas of interest throughout their career, will exhibit a number of change points in their publication records [96]. Additionally, change detection has been applied to communication graphs [7] and network traffic [87] by measuring the change in the eigen decompositions.

2.4 Methods

A common two-stage methodology was found among the papers reviewed. In the first stage, data-specific features are generated by applying a mapping from the domain-specific representation, graphs, to a common data representation, a vector of real numbers. A simple example is taking a single static graph as input and outputting its diameter. In the second stage, an anomaly detector is applied, taking the output from stage one, and optionally some historical data, and mapping it to a decision on the anomalousness of the data. Anomaly detectors, such as support vector machines (SVMs) [167] or the local outlier factor (LOF) algorithm [29], are general methodologies that are domain independent, since they operate on the common data representation that is output from stage one. Hence, stage two consists of identifying outlier points in a multi-dimensional space [4, 13, 14, 29, 133]. The two stages can be formally defined as follows:

\[
\begin{align*}
\text{Stage one} & \Rightarrow \Omega : D \rightarrow \mathbb{R}^d \\
\text{Stage two} & \Rightarrow f : \{[\mathbb{R}^d]^*, [\mathbb{R}^d]^t\} \rightarrow \{0, 1\}
\end{align*}
\]

where \( D \) is the domain-specific representation, \( d \) is the number of feature dimensions, \([\mathbb{R}^d]^*\) denotes the history of the data, and \([\mathbb{R}^d]^t\) denotes the current time point. Often times, stage one maps the domain-specific data to a single value, \( d = 1 \), such as the graph diameter example. Additionally, stage two can be generalized to mapping to \([0, 1]\) instead of \(\{0, 1\}\), representing the probability that the data are anomalous or a normalized outlier score assigned to the data.

This two stage process can be found in many other areas of anomaly or outlier detection, such as text mining and abnormal document detection [160], and computer vision [135]. However, in this chapter we restrict the scope to anomaly detection in dynamic graphs. The papers we surveyed are interested in finding the best mapping from dynamic graphs to a vector of real numbers, then applying existing anomaly detectors to identify the anomalies. Equivalently, they
are proposing methodologies for stage one, and applying existing methodologies for stage two. A complete list of the methods surveyed is provided in Tables A.2 and A.3.

2.4.1 Community Detection

Community based methods track the evolution of communities and their associated vertices in the graphs over time [168–170]. While the data-specific features generated by the methods discussed in this section are all derived using the community structure of the graphs, the various approaches differ in two main points: (i) in the aspects of the community structure they analyze, e.g., the connectivity within each community vs. how the individual vertices are assigned to the communities at each time step; and (ii) in the definitions of communities they use, e.g., soft communities, where each vertex has a probability of belonging to each community vs. hard, specifically disjoint, communities, where each vertex is placed into at most one community in the graph. Moreover, based on how the community evolution is viewed, it can be applied to the detection of different anomaly types. For example, a rapid expansion or contraction of a community could indicate that the specific subgraph for that community is undergoing drastic changes, whereas a drop in the number of communities by two corresponds to an abnormal event.

2.4.1.1 Vertex detection

Main Idea: A group of vertices that belong to the same community are expected to exhibit similar behavior. Intuitively this means that if at consecutive time steps one vertex in the community has a significant number of new edges added, the other vertices in the community would also have a significant number of new edges. If the rest of the vertices in the community did not have new edges added, the vertex that did is anomalous.

Based on this logic, using soft community matching and looking at each community individually, the average change in belongingness (the probability the vertex is part of the community) for each vertex can be found for consecutive time steps. Vertices whose change in belongingness is different from the average change of the vertices in the community are said to be evolutionary community outliers [76].

The changes in a vertex’s community belongingness may form a pattern over time, called a soft temporal pattern. In [152], a Non-Negative Matrix Factorization approach in combination with the Minimum Description Length principle are used to automatically detect vertex roles and build transition models. The community memberships are found slightly differently in [75], where Xmeans is used. However, in both cases, the patterns common across all of the vertices in the graph are extracted, then each vertex’s patterns are compared to the extracted ones. If a vertex’s patterns are not similar to any of the extracted common patterns, then the vertex
is anomalous. Later extended to static networks derived from heterogeneous data sources [74], the two step approach in [75] is modified to an alternating iterative approach in [76]. Instead of extracting patterns first and then identifying outliers, the patterns and outliers are found in an alternating fashion (pattern extraction $\rightarrow$ outlier detection $\rightarrow$ pattern extraction $\rightarrow$ $\cdots$) until the outliers discovered do not change on consecutive iterations. By alternating back and forth between pattern extraction and outlier detection, the algorithm accounts for the affect the outliers have on the communities discovered.

In [88], communities manifest in the form of corenets. Instead of defining the corenet (community) based solely on density, modularity, or hop distance (as egonets are [12]), the definition is based on the weighted paths between vertices. More formally, a vertex's corenet consists of itself and all the vertices within two hops that have a weighted path above a threshold value. If the edge weight between two vertices is considered the strength of their connection, then intuitively the vertices connected with higher weight edges should be considered part of the same community. Consequently, if a vertex has two neighbors removed, one connected with a high edge weight and the other connected with a low edge weight, then the removal of the vertex connected by the higher edge weight should have more of an impact. At each time step every vertex is first given an outlier score based on the change in its corenet, and the top $k$ outlier scores are then declared anomalous.

2.4.1.2 Subgraph detection

**Main Idea:** Instead of looking at individual vertices and their community belongingness, entire subgraphs that behave abnormally can be found by observing the behavior of communities themselves over time.

Six different types of community-based anomalies are proposed in [41]: shrink, grow, merge, split, born, and vanish. A graph at time $t$ is compared to the graphs at time $t + 1$ and $t - 1$ using graph and community representatives. The representatives reduce the computation required, providing seeding points for community enumeration. Communities are defined as maximal cliques, hence enumerating all communities is NP-hard, and overlapping communities are allowed. Graph representatives are the set of vertices that have appeared in time step $t$ in addition to $t + 1$, $t - 1$, or both; a community representative is a vertex that appears in the fewest number of the other communities. A set of rules are then derived based on these representatives that decide whether communities are anomalous, falling into one of the six defined classes, or normal. One additional type of community-based anomaly was proposed recently in [17]: comets, or communities that come and go, often periodically. The time-evolving graph is represented by a tensor (3-mode matrix), and the comets are detected by low rank tensor decomposition combined with the Minimum Description Length principle that allows for parameter-free community search.
Conversely, instead of finding changes, communities that are conserved, or stable, can be identified. Constructing multiple networks at each time step based on different information sources, communities can be conserved across time and networks. Networks that behave similarly can be grouped using clustering or prior knowledge. In [42], if a community is conserved across time steps and the networks within its group, but has no corresponding community in any other group of networks, then the community is defined as anomalous; two communities are considered corresponding communities if they have a certain percentage of their vertices in common.

Unlike [17, 41, 42] that consider only the structure of the network, in social networks there is often more information available. For example, in the Twitter user network, clusters can be found based on the content of tweets (edges), as well as the users (vertices) involved. If the fraction of the tweets (edges) added during the recent time window for a cluster is much larger than the fraction of tweets (edges) added anytime before the window, then this influx is declared as an evolution event for that cluster [3]. A cluster that experiences an evolution event is marked as an anomalous subgraph at the time when the evolution event occurs. Although the authors did not use labeled datasets in [17], the proposed algorithm can be applied to such data by appropriately incorporating the information in the tensor.

2.4.1.3 Change detection

**Main Idea:** Changes are detected by partitioning the streaming graphs into coherent segments based on the similarity of their partitionings (communities). The beginning of each segment represents a detected change.

The segments are found online by comparing the vertex partitioning of the newest graph to the partitioning found for the graphs in the current, growing, segment. Vertex partitioning can be achieved with many methods, but in [53] it is done using the relevance matrix computed by random walks with restarts and modularity maximization. When the partitioning of the new graph is much different from the current segment’s, a new segment begins, and the time point for the new graph is output as a detected change. The similarity of two partitions is computed as their Jaccard coefficient, and the similarity of two partitionings is the normalized sum of their partition similarities. A similar approach, GraphScope [163], based on the same idea but using the MDL principle to guide its partitioning and segmenting will be discussed in the following section.

2.4.2 Compression

The methods discussed in this section are all based on the minimum description length (MDL) principle [149]. MDL, and compression techniques based on MDL, exploit patterns and regularity in the data to achieve a compact graph representation [71]. Applying this principle to graphs is
done by viewing the adjacency matrix of a graph as a single binary string, flattened in row or
column major order. If the rows and columns of the matrix can be rearranged such that the
entropy of the binary string representation of the matrix is minimized, then the compression
cost (also known as encoding cost [147]) is minimized. The data-specific features are all derived
from the encoding cost of the graph or its specific substructures; hence, anomalies are then
defined as graphs or substructures (e.g., edges) that inhibit compressibility.

2.4.2.1 Edge detection

Main Idea: An edge is considered anomalous if the compression of a subgraph has higher encoding
cost when the edge is included than when it is omitted.

In [33], a two-step alternating iterative method is used for automatic partitioning of the
vertices. Vertex partitioning can be done by rearranging the rows and columns in the adjacency
matrix. In the first step, the vertices are separated into a fixed number of disjoint partitions
such that the encoding cost is minimized. The second step iteratively splits the partitions with
high entropy into two. Any vertex whose removal from the original partition would result in
a decrease in entropy is removed from that partition and placed into the new one. Once the
method has converged, meaning steps 1 and 2 are unable to find an improvement, the edges can
be given outlierness scores. The score for each edge is computed by comparing the encoding
cost of the matrix including the edge to the encoding cost if the edge is removed. Streaming
updates can be performed using the previous result as a seed for a new run of the algorithm,
thus avoiding full recomputations.

2.4.2.2 Change detection

Main Idea: Consecutive time steps that are very similar can be grouped together leading to
low compression cost. Increases in the compression cost mean that the new time step differs
significantly from the previous ones, and thus signifies a change.

Akin to [53], to detect changes in a graph stream, consecutive time steps that are similar
can be grouped into segments. When considering the next graph in the stream, it can either be
grouped with the graphs in the current segment, or it can be the beginning of a new segment.
The decision to start a new segment in GRAPHSCOPE [163] is made by comparing the encoding
cost of the current segment without the next graph to the encoding cost of the segment if the
next graph were included. If the vertex partitioning for the new graph is similar to the vertex
partitioning of the graphs in the current segment, then the encoding cost will not change much.
However, if the partitionings are quite different, the encoding cost would increase as a result of
the increase in entropy. Changes in the graph stream are the time points when a new segment
begins, and the algorithm is parameter-free.
2.4.3 Matrix/Tensor Decomposition

These techniques represent the set of graphs as a tensor, most easily thought of as a multi-dimensional array, and perform tensor factorization or dimensionality reduction. The most straightforward method for modeling a dynamic graph as a tensor is to create a dimension for each graph aspect of interest, e.g., a dimension for time, source vertices, destination vertices. For example, modeling the Enron email dataset can be done using a 4-mode tensor, with dimensions for sender, recipient, keyword, and date. The element \((i, j, k, l)\) is 1 if there exists an email that is sent from sender \(i\) to recipient \(j\) with keyword \(k\) on day \(l\), otherwise, it is 0.

Similar to compression techniques, decomposition techniques search for patterns or regularity in the data to exploit. All of the data-specific features generated by these methods are derived from the result of the decomposition of a matrix or tensor. Unlike the community methods, these typically take a more global view of the graphs, but are able to incorporate more information (attributes) via additional dimensions in a tensor. One of the most popular methods for matrices (2-mode tensors) is singular value decomposition (SVD) \([69]\), and for higher order tensors (\(\geq 3\) modes) is PARAFAC \([79]\), a generalization of SVD. The main differences among the decomposition based methods are whether they use a matrix or a higher order tensor, how the tensor is constructed (what information is stored), and the method of decomposition.

2.4.3.1 Vertex detection

**Main Idea:** Matrix decomposition is used to obtain activity vectors per vertex. A vertex is characterized as anomalous if its activity changes significantly between consecutive time steps.

Due to the computational complexity of performing principal component analysis \([90]\) (PCA) on the entire graph, it is computationally advantageous to apply it locally. The approach used in \([183]\) is to have each vertex maintain an edge correlation matrix \(M\), which has one row and column for each of its neighbors. The value of an entry in the matrix for vertex \(i\), \(M_{ij}(j,k)\), is the correlation between the weighted frequencies of the edges \((i,j)\) and \((i,k)\). The weighted frequencies are found using a decay function, where edges that occurred more recently have a higher weight. The largest eigenvalue and its corresponding vector obtained by performing PCA on \(M\) are summaries of the activity of the vertex and the correlation of its edges, respectively.

The time series formed by finding the changes in these values are used to compute a score for each vertex at each time step. Vertices that have a score above a threshold value are output as anomalies at that time.

2.4.3.2 Event detection

**Main Idea:** There are two main approaches: (a) Tensor decomposition approximates the original data in a reduced dimensionality, and the reconstruction error is an indicator of how well the
original data is approximated. Sub-tensors, slices, or individual fibers in the tensor that have high reconstruction error do not exhibit behavior typical of the surrounding data, and reveal anomalous vertices, subgraphs, or events. (b) Singular values and vectors, as well as eigenvalues and eigenvectors are tracked over time in order to detect significant changes that showcase anomalous vertices.

Using the reconstruction error as an indicator for anomalies has been employed for detecting times during molecular dynamics simulations where the collective motions suddenly change [138], finding frames in a video that are unlike the others [174], and identifying data that do not fit any concepts [22].

To address the intermediate blowup problem—when the input tensor and output tensors exceed memory capacity during the computation—Memory-Efficient Tucker (MET) decomposition was proposed [93], based on the original Tucker decomposition [175]. The Tucker decomposition approximates a higher order tensor using a smaller core tensor (thought of as a compressed version of the original tensor) and a matrix for every mode (dimension) of the original tensor. A variety of other tensor decompositions have been developed for offline, dynamic, and streaming analysis [162], in addition to static and sliding window based methods [166]. These extensions allow the method to operate on continuous graph streams as well as those with a fixed number of time points. Compact Matrix Decomposition (CMD) [164] computes a sparse low rank approximation of a given matrix. By applying CMD to every adjacency matrix in the stream, a time series of reconstruction values is created and used for event detection. Colibri [173] and PARCube [131] can be used in the same fashion and provide a large increase in efficiency. The PARAFAC decomposition has been shown to be effective at spotting anomalies in tensors as well [96].

Instead of finding the difference between the graph reconstructed from the approximation obtained by a decomposition, a probabilistic model can be used. The Chung-Lu random graph model [45] is used in [114]. Taking the difference between the real graph’s adjacency matrix and the expected graph’s forms a residual matrix. Anomalous time windows are found by performing SVD on the residual matrices—on which a linear ramp filter has been applied—and by analyzing the change in the top singular values. The responsible vertices are identified via inspection of the right singular vectors. More accurate graph models that also consider attributes are proposed in [115].

Going away from comparing an expected or approximate model of the graph to the real model to find the deviation, events can be identified via significant changes in the decomposed space. Specifically, by performing PCA on the data, the calculated eigenvectors can be separated into “normal” and “anomalous” sets by examining the values of the data projected onto each eigenvector. At each time step the eigenvectors are examined (via projection of the data) in descending order of their corresponding eigenvalues, and the first eigenvector to contain a
projected point outside 3 standard deviations of the rest of the values, and every eigenvector thereafter, constitute the anomalous set. The second step is then to project the data onto its normal and anomalous subspaces. Once this is done, events are detected when the modifications in the anomalous components from the previous time step to the current are above a threshold value [99]. Expanding on this method, joint sparse PCA and graph-guided joint sparse PCA were developed to localize the anomalies and identify the vertices responsible [89]. The responsible vertices are more easily identified by using a sparse set of components for the anomalous set. Vertices are given an anomaly score based on the values of their corresponding row in the abnormal subspace. As a result of the anomalous components being sparse, the vertices that are not anomalous receive a score of 0. Due to the popularity of PCA in traffic anomaly detection, a study was performed identifying and evaluating the main challenges of using PCA [146].

2.4.3.3 Change detection

Main Idea: The activity vector of a graph, \( \mathbf{u}(t) \), is the principal component, the left singular vector corresponding to the largest singular value obtained by performing SVD on the weighted adjacency matrix. A change point is when an activity vector is substantially different from the “normal activity” vector, which is derived from previous activity vectors.

The normal activity vector, \( \mathbf{r}(t-1) \), is the left singular vector obtained by performing SVD on the matrix formed by the activity vectors for the last \( W \) time steps. Each time point is given a score \( Z(t) = 1 - \mathbf{r}(t-1)^T \mathbf{u}(t) \), which is intuitively a score of how different the current activity is compared to normal, where a higher value is worse. Anomalies can be found online using a dynamic thresholding scheme, where time points with a score above the threshold are output as changes [86]. The vertices responsible are found by calculating the ratio of change between the normal and activity vectors. The vertices that correspond to the indexes with the largest change are labeled anomalous. Similar approaches have used the activity vector of a vertex-to-vertex feature correlation matrix [7], and a vertex-to-vertex correlation matrix based on the similarity between vertices neighbors [87].

2.4.4 Distance Measures

Using the notion of distance as a metric to measure change is natural and widely used [29, 40, 84, 130]. Two objects that have a small difference in a measured metric can be said to be similar. The metrics measured in graphs are typically structural features, such as the number of vertices. Once the summary metrics are found for each graph, the difference or similarity, which are inversely related, can be calculated. The variation in the algorithms lies in the metrics chosen to extract and compare, and the methods they use to determine the anomalous values and corresponding graphs.
2.4.4.1 Edge detection

**Main Idea:** If the evolution of some edge attribute (e.g., edge weight) differs from the “normal” evolution, then the corresponding edge is characterized as anomalous.

In [105], a dynamic road traffic network whose edge weights vary over time is studied. The similarity between the edges over time is computed using a decay function that weighs the more recent patterns higher. At each time step, an outlierness score is calculated for each edge based on the sum of the changes in similarity scores. Edges with the highest score, chosen using either a threshold value or top-k heuristic, are marked as anomalous at that time step.

Viewing each individual network in the sequence as a timestamped stream of edges, meaning the network does not have a fixed topology as the road traffic network did, the frequency and persistence of an edge can be measured and used as an indicator of its novelty. The persistence of an edge is how long it remains in the graph once it is added, and the frequency is how often it appears. In [1], set system discrepancy [38] is used to measure the persistence and frequency of the edges. When an edge arrives, its discrepancy is calculated and compared to the mean discrepancy value of the active edge set. If the weighted discrepancy of the edge is more than a threshold level greater than the mean discrepancy, the edge is declared novel (anomalous). Using the novel edges detected, a number of metrics can be calculated for various graph elements (e.g., vertices, edges, subgraphs). Individual graph elements can then be identified anomalous via comparison of the calculated metrics for that element with the overall distribution and change of the metric.

2.4.4.2 Subgraph detection

**Main Idea:** A subgraph with many “anomalous” edges is deemed anomalous.

Contrasting with [88] where the edge weights are considered the strength of a connection, in [119] the edge weights are considered as outlier scores, or how “anomalous” that edge is at that time. Every edge at every time step is given its own anomaly score, which is a function of the probability of seeing that particular edge weight on that particular edge given the distribution of weights for that edge over all the graphs in the series. Alternatively, the output of an existing method that assigns outlier scores for edges in a network [1, 33, 82, 105, 172] could be used as the input to this method. The latter approach allows [119] to be applied to any network that is capable of having outlier scores assigned to edges (e.g., weighted, directed, attributed). Once every edge is assigned an outlier score, Significant Anomalous Regions (SARs) – fixed subgraphs over a window of time – are mined from the sequence, analogous to the problem of finding the Heaviest Dynamic Subgraphs (HDS). An alternating iterative approach based on [27] – first finding an optimal time window for a fixed subgraph, then finding an optimal subgraph for the fixed time window, repeating until no improvement is found – is constructed to approximate
a solution to the NP-hard problem. This work was later extended to allow the subgraphs to smoothly evolve over time, where vertices can be added or removed between adjacent time steps [118]. A similar approach mines weighted frequent subgraphs in network traffic, where the edge weights correspond to the anomaly contribution of that edge [81].

2.4.4.3 Event detection

**Main Idea:** Provided a function \( f(G_i, G_j) \) that measures the distance between two graphs, a time series of distance values can be constructed by applying the function on consecutive time steps in the series. Anomalous values can then be extracted from this time series using a number of different heuristics, such as selecting the top \( k \) or using a moving average threshold.

Extracting features from the graphs is a common technique to create a summary of the graph in a few scalar values, its signature. *Local* features are specific to a single vertex and its egonet [12] (the subgraph induced by itself and its one-hop neighbors), such as the vertex or egonet degree. *Global* features are derived from the graph as a whole, such as the graph radius. The local features of every vertex in the graph can be agglomerated into a single vector, the signature vector, of values that describe the graph using the moments of distribution (such as mean and standard deviation) of the feature values. In [25], the similarity between two graphs is the Canberra distance, a weighted version of the \( L_1 \) distance, between the two signature vectors. A similar approach is used in [101] to detect abnormal times in traffic dispersion graphs. Instead of an agglomeration of local features, it extracts global features from each graph, introducing a \( dK-2 \) series [109, 153, 179] based distance metric, and any graph with a feature value above a threshold is anomalous.

As an alternative to extracting multiple features from the graph and using the signatures, the pairwise vertex affinity scores may be used. Pairwise vertex affinity scores are a measure of how much each vertex influences another vertex, and can be found using fast belief propagation [97]. In [58] the scores are calculated for two consecutive time steps, and the similarity between the two graphs is the rooted Euclidean distance (Matusita distance) between the score matrices. The changes in the vertex affinity score are shown to accurately reflect and scale with the level of impact of the changes. For example, removing an edge that connects two otherwise disconnected components, a “bridging edge,” results in a higher score than removing an edge that does not affect the overall structure of the graph. A moving threshold is set on the time series of similarity scores using quality control with individual moving range. The exponential weighted moving average has also been used as a way to dynamically set the threshold, tested on distribution features extracted from Wikipedia revision logs [16].

Complementary to feature similarity, one can look at the structural differences between graphs to identify the magnitude of change. These methods focus on the function that defines
the distance between graphs instead of finding the optimal features to use as summaries. Many
metrics have been developed and tested to quantify the differences between graphs. In [134],
ten different distance functions were evaluated on TCP/IP traffic graphs with known anomalies
(ground truth). Box-Jenkins autoregressive moving average (ARMA) modeling [28] was used to
create a “normal” model of the feature values, and the time points with residuals, calculated
as the difference between expected and real value, exceeding a threshold are flagged. The ten
distance functions tested were weight, maximum common subgraph (MCS) weight, MCS edge,
MCS vertex, graph edit distance, median graph edit distance, modality, diameter, entropy, and
spectral. Of these ten distance functions, the MCS based methods performed the best; however,
finding the MCS between two graphs is an NP-hard problem. More recently in [129], five different
distance scoring functions, all with a linear complexity, were tested on web graphs with specific
types of anomalies: missing subgraph, missing vertices, and connectivity change. The five scoring
functions were: vertex/edge overlap, vertex ranking, vector similarity, sequence similarity, and
signature similarity. The method that performed the best was the signature similarity, which
is done by extracting a signature vector from each graph and finding the distance between
them using SimHash [37]. Unlike the signature vectors discussed above that create summaries
of the entire graph in a few scalar values, the signature here includes every vertex and edge.
Specifically, the features used were each vertex and its PageRank value, and each edge \((u, v)\)
with a weight equal to \(u\)’s PageRank value divided by the number of outlinks from \(u\). Weighting
the vertices with their “quality,” measured by PageRank, ensures that the removal of a high
quality vertex will have more of an impact than the removal of a low quality vertex. A fixed
threshold was set to find graphs with abnormally low similarity scores.

Instead of finding structural difference between two consecutive graphs, events can be
detected using the time series of robustness values for each graph. Robustness is a measure
of the connectivity of the graph. A graph with a high robustness will retain the same general
structure and connectivity even when some vertices or edges are removed. Finding events is
then finding when the robustness value changes significantly [111]. A similar approach is given
in [65], but using a variant of the graph diameter. The graph diameter used is the average of all
vertex eccentricities (the eccentricity of a vertex \(v\) is the longest shortest path from \(v\) to any
other vertex), instead of the typical definition of using the maximum vertex eccentricity. Both
the time series of graph diameter changes and the graph diameters themselves are shown to be
effective methods for detecting events.

2.4.5 Probabilistic Models

With a foundation in probability theory, distributions, and scan statistics, these methods typically
construct a model of what is considered “normal” or expected, and flag deviations from this

model as anomalous. The type of model used, how it is constructed, what it is modeling, and the method for determining outliers is what differentiates these approaches.

2.4.5.1 Vertex detection

Main Idea: There are two main approaches: (a) Building scan statistics time series and detecting points that are several standard deviations away from the mean; (b) Vertex classification.

Scan statistics are often called “moving window analysis,” where the local maximum or minimum of a measured statistic is found in specific regions of the data. In a graph, a scan statistic can be considered as the maximum of a graph invariant feature, such as the number of edges, found for each vertex and its neighborhood in the graph. In [136] they use a variable “scale” for the measured statistic; each vertex has the number of edges in its 0, 1, and 2 step neighborhood measured. The local statistic for each vertex is then normalized using the mean and standard deviation of its recent values, and the scan statistic of a graph is the maximum of all of the normalized local statistics. Normalizing the values accounts for the history of each vertex, meaning the statistic for each vertex is relative to its own past instead of the values of the other vertices. This ensures that the largest change measured in the graph is not tied directly to the magnitude of change, but the ratio. Building a standardized time series of the scan statistic values, any value that is five standard deviations away from the mean of the series is considered an event (hence, events are detected as well). The vertex most responsible is identified as the one that was chosen for the scan statistic value for the entire graph.

Similar to the use of neighborhoods for scan statistics, the Markov Random Field Model (MRF) is used to uncover the states for vertices and infer the maximum likelihood assignment by a belief propagation algorithm, where a vertex is labeled based on the vertices it is connected to. In [128], anomalies (fraudsters) are uncovered in an online auction network by discovering bipartite cores, which are posited to be the interaction behavior between fraudsters and accomplices. It incrementally updates the model as new edges are added, taking advantage of the fact that an edge insertion or removal will affect only a local subgraph. In the propagation matrix a vertex can be in one of three states: fraudster, accomplice, or honest. Vertices that are assigned the label of fraudster are anomalous.

2.4.5.2 Edge detection

Main Idea: Communications (edges) are modeled using a counting process, and edges that deviate from the model by a statistically significant amount are flagged.

One way to model the relationship between vertices is considering the communication between them as a counting process. In [82], a Bayesian discrete time counting process is used to model the number of communications (edge weights) between vertices, and is continuously updated
as new graphs are considered. Predictive $p$-values, based on the learning of the distribution of the counts, are calculated for the edges of the newest observation and subsequently used for flagging anomalous vertex pairs (edges). Moreover, as new graphs are considered, both sequential – comparing the new graph to the model based on history – and retrospective analysis – adjusting the history based on the newest graphs – are performed. The retrospective analysis helps alleviate the insufficient data problem, when decisions are made early on with insufficient history to be accurate.

2.4.5.3 Subgraph detection

*Main Idea:* Fixed subgraphs (e.g., paths and stars), multigraphs, and cumulative graphs are used to construct models on the expected behaviors. Deviations from the models signify an anomalous subgraph.

To identify hacker behaviors in a network, [124] combines scan statistics with a Hidden Markov Model (HMM) for edge behavior. Unlike [136] that used neighborhoods, the local scan statistics are based on two graph shapes, the $k$-path and star. Comparing the scan statistics of a sliding window to its past values, and using an online thresholding system, local anomalies are identified. The local anomaly is the union of all subgraphs (representing the $k$-paths or stars) that led to statistically significant test statistics.

Another method to model a dynamic network, instead of using a series or stream of graphs, is to have a single multigraph where parallel edges correspond to communication between vertices at two different time steps. The initial multigraph is decomposed into telescoping subgraphs for each time window [52]. A telescoping subgraph (TSG) satisfies two conditions: (i) for any two edges that share a vertex, the edge that begins communication finishes communication last; (ii) there exists a vertex $r$, the root, that has no incoming edges and has a path to every vertex in the TSG. Telescoping subgraphs that have a low probability of appearing (according to their size and historic connection patterns), are output as anomalies.

Likewise, a cumulative graph, which includes every edge seen up until the current time step, is used in [171]. Edge weights in the graph are calculated using a decay function where more recent edges weigh more. A normal behavior for subgraphs is defined by identifying persistent patterns, here found by constructing a graph where each edge is weighted by the fraction of time it is in the graph and iteratively extracting the heaviest weight connected components. As the cumulative graph evolves the extracted subgraphs are monitored, comparing their current activity to the expected activity, which is based on the recent behavior.
### 2.4.5.4 Event detection

**Main Idea**: Deviations from the models of the graph likelihood or the distribution of the eigenvalues reveal when an event occurs.

Recently, a new reservoir sampling method was proposed in [5] to maintain structural summaries of a graph stream. The online sampling method manages multiple distinct partitionings of the network to provide a statistically significant summary. As a new graph is added to the stream, every edge has its likelihood calculated based on the edge generation models of the different partitions. Once the edge likelihoods have been found an global graph likelihood is calculated as the geometric mean of the edge likelihood values. A similar edge generation model approach was used in [85], where the probability that an edge exists between vertices $i$ and $j$ is stored in a matrix, $M(i,j)$. The estimated probabilities are calculated using expectation maximization, and subsequently used to give potential scores to every sender-recipient pair. Each sender-recipient tuple, which is an email from one sender to multiple recipients, then has its likelihood computed based on the estimated distribution of all possible sender-recipient tuples. A single score is calculated for each graph as the average of the log-likelihood scores. In both of these methods a single score is calculated for each graph based on the likelihoods of the individual edges within it, and the individual edges (or tuples) responsible are those that have the lowest likelihood.

Similar to estimating the distribution of possible sender-recipient tuples, the distribution of the eigenvalues and compressed eigen equations is calculated in [83]. Based on the assumption that each vertex has a time series of a vertex-local feature (e.g., summed edge weight), a vertex-to-vertex correlation matrix can be generated for each time step. The eigen equation of the correlation matrix is compressed by keeping the largest eigenvalues and a set of low dimension matrices (one matrix for each vertex). By learning the distributions of the eigenvalues and matrices, both events and the vertices responsible can be identified. When the eigenvalues deviate from the expected distribution an event has occurred, and the vertices whose matrices deviate from the matrix distribution [73] the most are considered responsible [83].

### 2.5 Discussion

With the wealth of algorithms discussed, the decision of which to use can be difficult. The choice depends on the types of graphs being analyzed (plain, attributed, directed, undirected, etc.), the desired types of anomalies, and the size of the graphs – influencing the allowed complexity, required parallelism, and whether streaming analysis is required. It is worth noting, however, that using multiple methods in parallel or in conjunction with one another may yield superior results compared to using a single method. Using publicly available datasets (Table A.4), each
algorithm can be applied to data from a variety of domains, comparing the found anomalies, their sensitivity, and their scalability.

While each method has its own advantages and disadvantages, for brevity we will give an overview of only those that have publicly available code (see Table A.5). We note that the available software listed is not comprehensive. We listed all software that was said to be available in their papers, as well as the software of the authors that responded to our email asking about the availability of their code. The methods are again partitioned by the types of anomalies they detect so that a fair qualitative comparison can be made. Note that a quantitative comparison among the available methods was impractical because the methods mostly detect different types of anomalies. Of the few intra-type comparisons that exist, coordinating the different outputs (e.g., anomalous vertices in the graphs vs. anomalous vertices in communities of the graphs) and attaining a valuable comparison would be unlikely.

Type 1 (vertices) methods [75, 76, 82]: The earliest work with available software is the online method by Heard et. al. [82], which models the frequency of the connections between vertices as a counting process and uses Bayesian learning and predictive $p$-values to identify anomalies. In addition to operating on graph streams, it leverages the sparsity of the network by only examining edges that appear in the graphs. A key advantage of this technique is that it performs both sequential analysis, where new graphs are analyzed using the history of the graph stream, in addition to retrospective analysis, where the history is updated based on the new graphs that arrive. One example of when this could be useful is in the initial stages of the analysis when very little history is available. When the algorithm terminates the output is very easy to interpret, labeling vertices/edges/subgraphs explicitly as anomalous or not. The flexibility of the final part of the algorithm, when analysis is performed on only the identified anomalous portions of the graph and their neighbors, opens many possibilities – anomalous community detection, identifying the most important or influential anomalous vertex, and many more.

Similar to [82], ECOOutlier [76] can be performed on graph streams. Each new graph will have a community detection algorithm run on it, and then the community matching and anomaly detection can be performed. However, it compares only two adjacent time points in the graph stream – the newly added graph and the one immediately preceding it. An interesting feature of this method is that it operates on the community belongingness matrices of a graph, and is therefore applicable to graphs of any type (plain, attributed, directed, etc.). The output from this method is not as straightforward as [82], at each time step providing a matrix that has the outlier score for each vertex in regards to each community. A related method, CTOutlier [75], has the same appeal of working on community belongingness matrices. A distinguishing factor is that CTOutlier operates on the sequence of belongingness matrices, requiring all of them to be in memory at once for a complete analysis. As a direct result of this, it is able to perform
community pattern extraction at an arbitrary scale, removing the restriction of comparing only adjacent time points. For long sequences the length of the patterns should be restricted to 2, as the number of potential patterns grows exponentially with the number of time steps. A single score for each vertex is output, representing how anomalous the community evolution for each vertex is over the entire sequence.

**Type 2 (edges) methods [82]:** Only the Bayesian learning technique by Heard et. al. [82] identifies anomalous edges explicitly. However, tensor methods, such as PARCube [131], can find the reconstruction error at arbitrary granularities, even on a per cell basis. Therefore, they are capable of identifying anomalous edges; however, the reconstruction error must then be maintained for each edge, instead of each tensor, or the user must provide individual edges of interest to examine. A more practical approach for using a tensor based method could be to identify the time points that are considered anomalous, and then examine the reconstruction error for the cells in those time points individually.

**Type 3 (subgraphs) methods [119]:** NetSpot [119] is designed to find optimal anomalous subgraphs. Its alternating optimization approach – fixing a subgraph and finding the optimal time window, then fixing the time window and finding the optimal subgraph within the window, and repeat – restricts the method to operating on an entire graph series at a time. The output from the algorithm is a set of highly anomalous subgraphs and their corresponding time windows, solving the problem of attribution and requiring no further analysis on the part of the user. Additionally, using their novel seed generation algorithm for mining heavy subgraphs, it is shown to scale linearly with the size of the network in terms of both vertices and time slices. While the authors assign outlier scores to the edges in the network in their own way, it would be interesting to utilize an algorithm specifically designed to assign outlier scores to edges in a network [1, 33, 82, 105]. Preprocessing the networks using an auxiliary edge scoring method allows a mapping from networks that do not fit the requirements of being plain, weighted, and undirected to networks that do.

As mentioned above, tensor methods are capable of identifying anomalous subgraphs using the reconstruction error for particular portions of the tensor. Without user guidance this is prohibitively expensive, as the number of possible subgraphs is exponential on the number of vertices in the graph. However, if the user is interested in a particular set of vertices, or a particular vertex and its neighbors, the error can be tracked for just that region of the graph.

**Type 4 (events/changes) methods [58, 131]:** DeltaCon [58] provides a graph similarity scoring function based on a number of desirable principles. It has been shown to effectively distinguish between, and correctly mark as more important, the removal of “bridging” edges and edges with higher weights, compared to edges that do not affect the overall structure if removed or have a low weight. Anomalies are found by finding the similarity between two adjacent graphs in the stream, flagging those that are sufficiently different from their immediate
neighbors as outliers. While not as robust as methods such as [82] that perform both sequential
and retrospective analysis, there is a commensurate decrease in the run time, scaling linearly
with the number of edges in the networks.

The tensor decomposition method ParCube [131] is the only method we discuss that is
capable of handling graphs that do not fit in memory. Moreover, it is parallelizable on a shared
memory machine and has been designed specifically for sparse tensors. As with other tensor
methods, it is able to handle any type of graphs with arbitrary attributes, as long as the
attributes can be reduced to a real number (e.g., edge labels being enumerated and mapping
them to integers). Using an online threshold, e.g., maintaining the mean and standard deviation
of the reconstruction error and setting the threshold to two standard deviations above the mean,
ParCube can be applied to graph streams.

2.6 Conclusion

In this chapter, we hoped to bridge the gap between the increasing interest in anomaly detection
in dynamic graphs and the extensive literature available. A common high level theoretical
formulation was found among the methods surveyed. Within this formulation, all of the methods
are concerned with exploring various mappings from graphs to real numbers to best identify
anomalous behavior using existing anomaly detectors. Although a myriad of approaches have
been proposed and explored, they all aim to identify one or more of the four key anomalies
we have outlined for dynamic networks: vertices (Figure 2.1), edges (Figure 2.3.2), subgraphs
(Figure 2.3), and events or changes (Figure 2.4). In conjunction with the types of anomalies
identified, we partition the methods into five main categories derived from the intuition behind
each: communities, compression, decomposition, distance, and probabilistic model based.

From this extensive analysis we have identified several open challenges and gaps in the
existing literature, the most pertinent of which follow. First, a major concern is the narrow
focus of graph models used, e.g. lacking attributes, constraining them to be sequences of graph
objects, lack of consideration for high order relationships. The simpler the graph model used,
the simpler the represented system must be, or the more information you lose about the
system. Second, many algorithms have a very high associated cost, in terms of both time and
space complexity. Real-time anomaly detection relies on efficient algorithms that are capable
of processing streaming data (as opposed to batched data, such as graph sequences). Finally,
evaluation of techniques. Many papers, our own future chapters included, lack ground-truth
data to evaluate methodologies with. Consequently, synthetic datasets, real-world datasets with
synthetic anomalies injected into it, or domain expert analysis (case studies) must be used for
evaluation.

The field of anomaly detection in dynamic networks is relatively young and is rapidly growing
in popularity, as indicated by the number of papers published in the past five years. A concise summary of the methods discussed in this survey is given in Tables A.2 and A.3, showing the type of anomalies detected, year of publication, and time complexity (notation defined in Table A.1). A list of papers that have made their code available to the public is given in Table A.5, and many publicly accessible datasets for testing are listed in Table A.4.
3.1 Introduction

As shown in Chapter 2, anomaly detection over dynamic graphs has been a topic of great interest recently given its many applications [140]. Typically, these dynamic graphs are broken into a stream, or sequence, of static graphs, and analysis is performed by comparing adjacent graphs in the stream [58, 134, 163]. However, there are several limitations to performing outlier detection in this manner. First, it requires that entire graphs be stored either in main memory or offline for analysis, which is infeasible given the large universe from which edges and vertices are drawn from and the rate at which the graphs change. For example, networks such as Twitter or Facebook can have on the order of $10^6$–$10^9$ active users (meaning $10^{12}$–$10^{18}$ possible unique edges), with Twitter processing 500 million tweets per day in 2013\(^1\) and 293,000 status updates every 60 seconds on Facebook\(^2\). Second, much or all of the historical information of the stream is lost, as only the most recent graphs in the stream are considered. Third, global comparisons of the graphs often precludes the possibility of performing attribution for the changes in the graph. In response to these limitations, we propose a transition to outlier detection in the context of

\(^1\)http://www.internetlivestats.com/twitter-statistics/
edge streams, where objects in the stream are individual edges from the graph instead of full graph objects.

As edge stream mining is a more fine-grained analysis of the changes, we hypothesize it is more much amenable to attribution. However, addressing memory constraints and historical information loss issues is still a challenging problem. Two common approaches for addressing these issues in stream mining are sampling and sketches [2]. Sampling attempts to retain the salient portions of data from the stream for analysis, while forgetting (not storing) the rest. The main challenge is how to determine what information is important in a single pass, and how to remain agnostic to the order in which the data is seen. Conversely, sketches maintain a compressed summary of the entire data stream. In this chapter, we propose a model for calculating an outlier score for each edge in a continuous edge stream, and we utilize the Count-Min (CM) sketch [48] to approximate the structural properties of the stream that are relevant to this model. Using this approximation, we are able to prove probabilistic error bounds on our edge scores.

We propose, to the best of our knowledge, the first method for identifying outliers in an edge stream. The major contributions of this chapter can be summarized as follows.

1. **Scalable Streaming Model for Outlier Detection.** We describe a data driven model for scoring edges in a continuous stream (Section 3.2). We propose a sketch-based approach that uses a constant amount of space that is independent of the size of both the graph and the stream itself, and performs updates and outlier analysis in constant time (Section 3.3, Section 3.4).

2. **Theoretical Bounds.** We show theoretical probabilistic bounds for the error of the estimations of our sketch-based approach (Section 3.3).

3. **Extensive Experiments.** Extensive experiments are performed to test scalability, measure the true error resulting from approximation, empirically test the effects of approximation on identifying injected ground-truth outliers, and validate the models ability to detect outliers in real-world data (Section 3.5).

### 3.2 Problem Statement and Model

We informally define the problem we address as follows.

**Problem 1. (Outlier detection in edge streams)**

**Given:** A continuous edge stream $\mathcal{E} = \langle(u, v), \ldots \rangle$, where $u$ and $v$ are the vertices the edge is incident upon.

**Find:** Edges in the stream that deviate from the expected structural patterns.
It is important to note the distinction between this formulation and the problem of link prediction. In link prediction, outliers are when edges that are expected to occur do not, while we define outliers as edges that are not expected to occur but do. For example, edges that occur in the stream that bridge two previously disconnected components of the graph would be an outlier by our problem definition, but not in the link prediction problem (see Section 3.1 for real-world examples). To this end, we propose a data-driven structural model that encapsulates both global edge characteristics and vertex-level egocentric behaviors.

3.2.1 Data Model

Before discussing the structural model, it is essential to understand the type of data we will be modeling. Every edge in the stream is undirected\(^3\), unweighted/unlabeled, and represents the insertion of a single edge into the graph. This is a special case of the “cash register model” \cite{122} for streams. \(E\) represents a multigraph, therefore, the number of times an edge may occur in the stream is unrestricted. Similarly, no \textit{a priori} knowledge on the total number of vertices in the graph is required, and the total vertex set may (and likely will) grow over time. Every vertex \(v\) is assumed to be represented by a unique label (e.g., usernames, IP addresses), \(l(v)\), thus \(l(u) = l(v)\) iff \(u = v\). Edges are labeled as the concatenation of the two vertices they are incident upon, thus the label of \(e = (u, v)\) is \(l(e) = l(u) \oplus l(v)\), where \(\oplus\) is the concatenation operator.

3.2.2 Edge Scoring and Outlier Detection

Given this data model, the goal is to identify edges that appear in the stream that deviate from the historical patterns of the graph, and thus the expected behavior of the incoming stream. Every incoming edge is scored based on historical evidence as well as connectivity patterns of the vertices it is incident upon. The score of an edge scales proportionally with the extent to which it conforms to the expected behavior, low scores representing outliers and high scores representing “normal” edges. This scoring, and consequently the labeling of an edge as an outlier or not, must be done in real-time. Moreover, given the vast size of the networks and the rate at which updates occur, the analysis should only require space that is sublinear, or independent of, the size of the network and stream. To achieve this, we propose a sketch-based approach for probabilistically modeling the edge stream with provable error bounds (see Section 3.3).

However, for simplicity and clarity when describing the high level outlier detection model, in this section, we assume that the stream is being stored exactly and explicitly using an adjacency matrix \(A\).

The structural properties of the edge stream will be modeled using a weighted graph, where every incoming edge \((u, v) \in E\) increments the corresponding edge weight \(A_{uv}\) by 1. Thus, \(^3\)Our model is easily generalized to directed edges.
if \((u, v)\) has occurred in \(E\) three times, \(A_{uv} = 3\). As we consider the undirected case, \(A\) is symmetric, \(A_{uv} = A_{vu}\). We define the weight of a vertex as the sum of all of its edge weights, \(w(u) = \sum_v A_{uv}\), and, abusing notation, we define the weight of an edge as \(w(u, v) = A_{uv}\). For a given vertex \(u\), its neighborhood is defined as the set of all vertices which are adjacent to it, \(N(u) = \{v \mid (u, v) \in E\} = \{v \mid A_{uv} > 0\}\).

### 3.2.2.1 Evidence-Based Scoring

One of the strongest indicators of an edge occurring in the stream is its historical presence. As discussed in [8], two dominating properties of dynamic real-world networks are their power law degree distributions and the self-similarity of edge additions. Intuitively, this equates to edges that have occurred more often having a higher probability of occurring again. This intuition leads us to a simple sample-based scoring function.

**Definition 3.1 (Sample Score).** The sample score of an edge \((u, v)\), \(s^s_{uv}\), occurring in the edge stream \(E\) is proportional to the weight of the existing edge between \(u\) and \(v\) compared to the total weight of the graph.

\[
s^s_{uv} = \frac{A_{uv}}{\sum_i \sum_{j > i} A_{ij}} = \frac{A_{uv}}{|E|} \quad (3.1)
\]

However, sample-based scoring does not account for the sparsity of real-world networks, where most vertex pairs are not connected. To smooth the sample-based scoring, we introduce a second scoring function. We use a normalized version of the preferential attachment score, commonly used in the link prediction community [106, 108] and made popular by the Barabási-Albert graph generation model [21]. This non-sample-based score scales proportionally with the degree of the two vertices it is incident upon, meaning edges that have never been seen before can be assigned a non-zero score. Normalizing it ensures that every score is mapped into the range \([0, 1]\), like the sample score is.

**Definition 3.2 (Preferential Attachment Score).** The preferential attachment score of an edge \((u, v)\), \(s^p_{uv}\), occurring in the edge stream \(E\) is proportional to the product of the weights of \(u\) and \(v\). Let \(||A_u|| = \sum_v A_{uv}\).

\[
s^p_{uv} = \frac{||A_u|| \cdot ||A_v||}{|E|^2} \quad (3.2)
\]

Note that this is a monotonically decreasing function for each edge. As more edges are seen in the stream, and the graph model grows more robust, the data-driven sample score begins to dominate over this smoothing score.
Figure 3.1 Example graph and table illustrating the scores each edge incident on vertex 2 would receive if it were to arrive next in the stream. For calculations, \( w(0, 2) = 3 \), all other edge weights are 1, \( \alpha + \beta = 0.75 \), and \( \gamma = 0.25 \). Note how the homophily score complements the evidence-based scores.

3.2.2.2 Neighborhood-Based

Evidence-based scoring functions provide accurate data-driven metrics for assessing the degree to which an edge fits the global expectation. Nevertheless, they fail to capture the edges fit when considering the local surrounding structure. Equivalently, they capture only direct connections, and fail to consider the indirect connections of vertices. For example, consider vertex 2 in Figure 3.1. Vertices 0 and 2 are connected by a heavily weighted edge, indicating that \((0, 2)\) has been seen in the stream numerous times. Therefore, it is expected that there will be more edges between them in the future. Conversely, vertices 1 and 2 have no edge between them, indicating that \((1, 2)\) has not been in the stream. Would it be strange then for an edge connecting them to appear? Using only sample-based scoring – yes. However, given the number of shared neighbors, it should not be surprising. The expectation that two vertices with many common neighbors will interact is a manifestation of the phenomenon known as homophily. In general, homophily is the tendency for objects to interact with other objects that are similar to themselves [112]. We derive a metric to quantify the similarity between two vertices by extending a simple metric commonly used in the link prediction literature. Given two vertices, the Jaccard Similarity Coefficient of their neighborhoods is a simple yet effective way to express the probability of having an edge between them. However, this metric treats all neighbors equally, ignoring the weights of their connections. Therefore, we extend this metric by proposing a weighted neighborhood intersection and an appropriate normalization.

Definition 3.3 (Homophily Score). The homophily score of an edge \((u, v)\), \(s_{uv}^h\), occurring in the
edge stream $\mathcal{E}$ is proportional to the weight of the intersection of the neighborhoods of $u$ and $v$.

$$s_{uv}^h = \frac{\sum_{k \in N(u) \cap N(v)} (A_{uk} + A_{vk})}{||A_u|| + ||A_v||}$$

(3.3)

Using Eqn. 3.3, not only is the size of the intersection considered, but the proportional weight as well. Moreover, the direct connection between the vertices incident on the edge is not included, removing the potential overlap with the sample score. As desired, given two vertex pairs with equal neighborhood intersection sizes, the pair with a higher percentage of their summed vertex weights on edges incident on vertices in the intersection will have a higher homophily score.

### 3.2.2.3 Total Score

Every edge is mapped to a single score, indicating how well it is in accordance with the expected structural behavior. We define the total score for an edge as the linear combination of the three above-mentioned scoring functions.

**Definition 3.4 (Total Edge Score).** The total score of an edge $(u, v)$, $s_{uv}$, occurring in the edge stream $\mathcal{E}$ is a linear combination of the evidence- and neighborhood-based scoring functions.

$$s_{uv} = \alpha s_{uv}^e + \beta s_{uv}^p + \gamma s_{uv}^h$$

where $\alpha + \beta + \gamma = 1$.

The coefficients $\alpha$, $\beta$, and $\gamma$ serve as weights between the scores, giving applications the flexibility to value, for example, the sample score over the homophily score. As all three scoring functions map to a real number in $[0,1]$, the total score is also in the range $[0,1]$. See Figure 3.1 for an example graph with all the scores shown.

### 3.2.2.4 Outlier Detection

By mapping every edge to a real-valued number, we have shifted the domain for outlier detection from graphs to time series. Outlier detection in time series data is a well-studied domain [84]. In this work, we choose a simple thresholding approach, but any approach that can evaluate a new data point in constant time is suitable. We can now restate Problem 1 more formally. Given a continuous edge stream $\mathcal{E}$, and a threshold score $\tau$, identify all outlier edges $(u, v) \in \mathcal{E}$ such that $s_{uv} < \tau$.

### 3.3 Model Approximation

Explicit and exact storage of the network structure for on or offline analysis is infeasible for streams at even a moderate scale. In [5], the authors introduce a structural reservoir sampling
Table 3.1 Notation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>Error factor for query response from sketch.</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Probability of count-query response from the sketch being within a factor of $\varepsilon$ is at least $1 - \delta$.</td>
</tr>
<tr>
<td>$w$</td>
<td>Width of a sketch. $w = \lceil \frac{\varepsilon}{\delta} \rceil$.</td>
</tr>
<tr>
<td>$d$</td>
<td>Depth of a sketch. $d = \lceil \ln \frac{1}{\delta} \rceil$.</td>
</tr>
<tr>
<td>$w(u, v)$</td>
<td>Weight of the edge between vertices $u$ and $v$. Equal to the number of times $(u, v)$ has appeared in the stream.</td>
</tr>
<tr>
<td>$N(u)$</td>
<td>Neighborhood of $u$. $N(u) = {v \mid (u, v) \in E}$.</td>
</tr>
<tr>
<td>$N_{uv}$</td>
<td>Sketch for vertex $u$ (used for estimating the homophily score).</td>
</tr>
<tr>
<td>$w(u)$</td>
<td>Number of cells that have a nonzero value in both $N_u$ and $N_v$.</td>
</tr>
<tr>
<td>$w'(u)$</td>
<td>Weight of a vertex $u$, defined as the sum of the weight of its edges. $w'(u) = \sum_{k \in N(u)} w(u, k)$.</td>
</tr>
<tr>
<td>$</td>
<td>E</td>
</tr>
<tr>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td>$s_{uv}$</td>
<td>Preferential attachment score of edge $(u, v)$. Defined by Eqn. 3.2.</td>
</tr>
<tr>
<td>$s_{hw}$</td>
<td>Homophily score of edge $(u, v)$. Defined by Eqn. 3.3.</td>
</tr>
<tr>
<td>$s_{u}$</td>
<td>Total score of an edge $(u, v)$. Defined by Eqn. 3.4.</td>
</tr>
<tr>
<td>$h_i(u)$</td>
<td>Value of the $i$th hash function applied to $u$.</td>
</tr>
</tbody>
</table>

A technique to heuristically store sub-structures in the graph to facilitate analysis on multiple samples of the graph. We propose a complementary method that uses a probabilistic data structure, known as the Count-Min (CM) sketch, to store an approximation of the complete stream. Sketch-based techniques are commonly used in streaming and time series scenarios to maintain important statistics of the overall stream in a greatly reduced space [2]. We will focus on the Count-Min sketch [48], and how it can be used to store the required edge frequencies, weights, and vertex neighborhood information for our model. In this section, we describe how CM sketches are used, and we prove probabilistic guarantees for the accuracy of the scoring functions when using the sketch-based model. Full notation is given in Table 3.1.

### 3.3.1 The CountMin Sketch

A CM sketch is a two-dimensional array of width $w$ and depth $d$ that is used for storing approximations of data stream frequencies. Frequency queries on CM sketches have known accuracy guarantees that are characterized by the parameters $\varepsilon$ and $\delta$, which dictate the error of the approximation and the probability of the approximation being within that error, respectively. More concretely, if $a$ is a vector with the true counts of the objects seen in the stream, then a CM sketch of width $w = \lceil \frac{\varepsilon}{\delta} \rceil$ and depth $d = \lceil \log \frac{1}{\delta} \rceil$ will give an object count within a factor of $\varepsilon ||a||$ with probability at least $1 - \delta$ [48].

As objects arrive in the stream, $d$ pairwise independent hash functions$^4$ are applied to the object’s unique identifier, and the counts at the corresponding cells in the sketch are updated.

$^4$By pairwise independent, we mean that for a given hash function $h_i$, $\mathbb{P}[h_i(x) = h_i(y) \mid x \neq y] = \frac{1}{w}$. 

41
Figure 3.2 How we use a CM sketch for approximating edge weights.

The error of the approximation comes from hash collisions since the width is much smaller than the total number of unique objects in the stream. It is important to note that every frequency approximation, from each of the \( d \) rows, will be at least as large as the true count. As a result, the best estimate for the true count of an object is the minimum of the \( d \) approximations.

### 3.3.2 Sample Score

A single sketch is required for estimating the sample score. Recall that every edge has a unique identifier that is the concatenation of the labels of the two incident vertices, \( l(e) = l(u) \oplus l(v) \). Thus, an estimation for the number of times every edge has appeared in the stream can be stored. As the number of edges seen in the stream is known exactly, the error is only introduced in the numerator of the scoring function. Using this, we can derive the following probabilistic error bound for the sample score of an edge.

**Lemma 3.1.** Using a Count-Min sketch of width \( w = \lceil \frac{e}{\varepsilon} \rceil \) and depth \( d = \lceil \ln \frac{1}{\delta} \rceil \), the sample score of an edge \( (u, v) \) is bound by the following inequality with probability at least \( 1 - \delta \).

\[
s_{uv}^s \leq s_{uv}^{se} \leq s_{uv}^s + \varepsilon
\]

**Proof. Lower bound:** The exact sample score of an edge \((u, v)\) is defined as \( s_{uv}^s = w(u, v)/|\mathcal{E}| \). The estimated sample score is then given by the equation \( s_{uv}^{se} = w_e(u, v)/|\mathcal{E}| \), where \( w_e(u, v) \) is the estimated weight of the edge using the Count-min sketch. As the estimate of the sketch is guaranteed to be greater than or equal to the true value, clearly, \( s_{uv}^s \leq s_{uv}^{se} \).

**Upper bound:** To prove the upper bound, it is sufficient to show that the following inequality holds with probability at least \( 1 - \delta \).

\[
w_e(u, v) \leq w(u, v) + |\mathcal{E}| \varepsilon
\] (3.4)
For a single hash function, the estimated value of an edge weight is equal to \(w_e(u, v) = w(u, v) + X_i\), where \(i\) is the index that \((u, v)\) hashes to and \(X_i\) is a variable accounting for the error produced by hash collisions in cell \(i\). Assuming a pairwise independent hash function, the expected value of the error for any given edge weight is \(E[X_i] \leq |\mathcal{E}|/w = m_i \varepsilon / e\). By the Markov inequality,

\[
P[w_e(u, v) > w(u, v) + |\mathcal{E}| \varepsilon] = P[X_i > w(u, v) + |\mathcal{E}| \varepsilon] \\
= P[X_i > eE[X_i]] < \frac{E[X_i]}{eE[X_i]} = e^{-1} < \delta
\]

From this, we can see that for \(d\) independent hash functions, \(P[w_e(u, v) > w(u, v) + |\mathcal{E}| \varepsilon] < e^{-d} \leq \delta\). Thus, Eqn. 3.4 holds with probability at least \(1 - \delta\).

This result implies that a very good estimation is possible using only a few megabytes of memory. For example, letting \(\varepsilon = .00001\) and \(\delta = .001\), only 7.26 megabytes (assuming 32-bit counters) are needed to give a score within .00001 of the true value with probability at least .999.

### 3.3.3 Preferential Attachment Score

The preferential attachment score is estimated in a very similar manner to the sample score. A single (different) CM sketch will be used to estimate the weight of every vertex in the graph. Again, the denominator is known exactly, resulting in the error being from the numerator only. Unlike the sample score, the error is multiplicative since the sketch is used to get the weight of each vertex of the given edge. Multiplicative error withstanding, the following bound is derived for the preferential attachment score.

**Lemma 3.2.** Using a Count-Min sketch of width \(w = \lceil \varepsilon / \delta \rceil\) and depth \(d = \lceil \ln 1 / \delta \rceil\), the preferential attachment score of an edge \((u, v)\) is bound by the following inequality with probability at least \(1 - \delta\).

\[
s_{uv}^p \leq s_{uv}^{pe} \leq s_{uv}^p + \varepsilon
\]

**Proof.** **Lower bound:** The exact preferential attachment score of an edge is defined as \(s_{uv}^p = w(u)w(v)/|\mathcal{E}|^2\) for some edge \((u, v)\). The estimated preferential attachment score is then given by the equation \(s_{uv}^{pe} = w_e(u)w_e(v)/|\mathcal{E}|^2\), where \(w_e(u)\) is the estimated score (using the Count-min sketch) for vertex \(u\). As the estimates of the sketch are guaranteed to be greater than or equal to the true value, clearly, \(s_{uv}^p \leq s_{uv}^{pe}\).
**Upper bound:** To prove the upper bound, it is sufficient to show that $w_e(u)w_e(v) - w(u)w(v) \leq |E|^2 \varepsilon$ with probability at least $1 - \delta$. Let $h(v)$ denote the hash function $h$ applied to vertex $v$. Observe that, for a single hash function,

$$w_e(u)w_e(v) = \left( w(u) + \sum_{p \neq u, h(p) = h(u)} w(p) \right) \left( w(v) + \sum_{p \neq v, h(p) = h(v)} w(p) \right).$$

Then, assuming a pairwise independent hash function,

$$\mathbb{E}[w_e(u)w_e(v) - w(u)w(v)] = \left( w(u) + \frac{1}{w} \sum_{p \neq u} w(p) \right) \left( w(v) + \frac{1}{w} \sum_{p \neq v} w(p) \right) - w(u)w(v) = \frac{1}{w} \left( w(u) \sum_{p \neq v} w(p) + w(v) \sum_{p \neq u} w(p) + \frac{1}{w} \sum_{p \neq u} \sum_{q \neq v} w(p)w(q) \right) \leq \frac{|E|^2}{w} = \frac{|E|^2 \varepsilon}{e}.$$

By the Markov inequality, we know that $\mathbb{P}[w_e(u)w_e(v) - w(u)w(v) > |E|^2 \varepsilon] < e^{-1}$. Hence, for $d$ independent hash functions, $\mathbb{P}[w_e(u)w_e(v) - w(u)w(v) > |E|^2 \varepsilon] < e^{-d} \leq \delta$. 

**3.3.4 Homophily Score**

To estimate the homophily score for an edge, we must somehow maintain approximate neighborhood information for every vertex in the graph. To estimate the neighborhood of a vertex, one option would be to use the sketch for the sample score (i.e., edge weights). Given a vertex $u$, an approximate neighborhood could then be found by iterating over all edges $(u, v)$; if the weight of an edge is greater than zero, then $v$ is considered part of the neighborhood of $u$. Formally we would say $N(u) = \{ v \mid v \in V \land w(u, v) > 0 \}$. However, with no information of the existing edges being stored explicitly, all $O(|V|)$ potential neighbors would have to be checked. In a streaming environment, this is unacceptable. Instead, we propose two sketch-based solutions: one with a provable probabilistic accuracy guarantee and one that is a heuristic based on well-known real-world network properties (the effectiveness of each solution is evaluated in Section 3.5).

**3.3.4.1 Individual Sketches Approximation**

Our first approach provides a provable probabilistic error bound. We use a (new) third sketch but, instead of a traditional two-dimensional sketch, we extend the sketch into an additional dimension. Thus, our new sketch is of size $w \times d \times l$, where length $l$ is the new dimension. At any given point in time, $l$ is equal to the number of unique vertices seen in the stream. Equivalently, every time a new vertex is seen, an additional sketch is allocated for it. With each vertex having
its own sketch, the width can be substantially smaller than the width of the sketches for the vertex weights and the edge weights. Real-world networks commonly follow a power law degree distribution [21], meaning the majority of vertices have a very small neighborhood. In addition to implying that the sketches can be smaller, it implies that it is rare that two vertices that do not already belong to the same small cluster will share a new edge. This is important because smaller sketches equate to mapping the same potential number of vertices ($|V|$) into a much smaller range, increasing the probability of collisions. The reduced number of edges between distant neighbors results in a decreased amount of error. In the following proof this manifests in the multiplier on the error term.

**Lemma 3.3.** Assume every vertex has their own sketch to track the weight of the edges between themselves and every vertex in their neighborhood. Given two vertices, $u$ and $v$, and their corresponding sketches, $N_u$ and $N_v$, each of width $w = \lceil \epsilon \rceil$ and depth $d = \lceil \ln \frac{1}{\delta} \rceil$, the homophily score of $(u, v)$ is bound by the following inequality with probability at least $1 - \delta$,

$$s^h_{uv} \leq s^{he}_{uv} \leq s^h_{uv} + \varepsilon N_{uv}$$

where $N_{uv}$ is the number of cells that have a nonzero value in both $N_u$ and $N_v$.

**Proof.** Lower bound: The exact and estimated homophily scores defined, respectively, as:

$$s^h_{uv} = \frac{\sum_{k \in N(u) \cap N(v)} (w(u, k) + w(v, k))}{\sum_{k \in N(u)} w(u, k) + \sum_{k \in N(v)} w(v, k)},$$

and

$$s^{he}_{uv} = \frac{\sum_{k \in N(u) \cap N(v)} (w_e(u, k) + w_e(v, k))}{\sum_{k \in N(u)} w_e(u, k) + \sum_{k \in N(v)} w_e(v, k)}$$

Notice that the denominators in the above equations are equivalent because the sum of any row in the sketch for a vertex is the exact weight of that vertex; namely, $||N_u|| = \sum_{k \in N(u)} w(u, k)$. As a result, we get the following:

$$s^h_{uv} = \frac{\sum_{k \in N(u) \cap N(v)} (w(u, k) + w(v, k))}{||N_u|| + ||N_v||},$$

and

$$s^{he}_{uv} = \frac{\sum_{k \in N(u) \cap N(v)} (w_e(u, k) + w_e(v, k))}{||N_u|| + ||N_v||}$$

Assuming every vertex’s sketch uses the same set of hash functions, the estimated weight of the intersection can be broken into two components, the true weight of the intersection plus the
error introduced by hash collisions.

\[ \sum_{k \in N(u) \cap N(v)} (w_e(u, k) + w_e(v, k)) = \sum_{k \in N(u) \cap N(v)} (w(u, k) + w(v, k)) + \sum_{q \in N(u) \cap N(v), \exists p \in N(u) \cap N(v), h(p) = h(q)} (w(u, q) + w(v, q)) \]  \hspace{1cm} (3.5)

It is clear from this that the estimated score will always be at least the non-estimated score.

**Upper bound:** We will show that \( s_{uv}^{he} \leq s_{uv}^h \) holds with probability at least \( 1 - \delta \) by showing that the following inequality holds with probability at least \( 1 - \delta \). It is clear that if we can show that

\[ \sum_{k \in N(u) \cap N(v)} (w_e(u, k) + w_e(v, k)) - \sum_{k \in N(u) \cap N(v)} (w(u, k) + w(v, k)) \leq N\varepsilon(||N_u|| + ||N_v||). \]  \hspace{1cm} (3.6)

holds with probability at least \( 1 - \delta \), then Eqn. 3.3 does as well.

Let \( X \) be a variable defined as the left hand side of Eqn. 3.6 and \( p = \mathbb{P}[\exists p \in N(u) \cap N(v), h(p) = h(q)] \). Substituting Eqn. 3.5 into Eqn. 3.6 and simplifying we get:

\[
\mathbb{E}[X] = \sum_{q \notin N(u) \cap N(v)} p(w(u, q) + w(v, q)) \\
\leq \sum_{q \notin N(u) \cap N(v)} \frac{N_{uv}}{w} (w(u, q) + w(v, q)) \\
\leq \frac{N_{uv}}{w} (||N_u|| + ||N_v||) = \frac{\varepsilon N_{uv}}{e} (||N_u|| + ||N_v||)
\]

By the Markov inequality, \( \mathbb{P}[X > N_{uv}\varepsilon(||N_u|| + ||N_v||)] \leq e^{-1} \). Generalizing to \( d \) different hash functions gives the desired result.

### 3.3.4.2 Sketch Sharing Heuristic

While providing a provable error bound, allocating a sketch (however small) for each vertex means the required space scales linearly with the number of vertices. However, recalling the sparsity of real-world networks, their degree distributions, and the tendency towards having small cohesive substructures (communities), we can reduce the required space, by allowing vertices to share neighborhood sketches. Again, we will use a (new) third sketch that is extended into three dimensions. However, this time, the length will be a fixed constant that is independent of the number of vertices seen.

Clearly, for any assignment of \( |V| \) vertices to \( l < |V| \) sketches, at least \( \lceil |V|/l \rceil \) vertices will share a sketch. To remove any potential bias and dependency on ordering, we randomly assign the vertices to a sketch by hashing the label of the vertex to a number in \([1, l]\). Moreover, random assignments minimize the error by keeping the expected number of vertices per sketch at \( \lceil |V|/l \rceil \). Again we assume pairwise independence in the hash function. From a given sketch’s perspective, each vertex assignment is a Bernoulli Trial with success \( p = \frac{1}{l} \), yielding a Binomial
Distribution described by \( n = |V| \) and \( p = \frac{1}{7} \), which has an expected value of \( \frac{|V|}{7} \). Even with a fixed length, the expected error can be reduced by modifying the width of the sketch, as the error is a function of the number of hash collisions, and the expected number of collisions is inversely proportional to the width of the sketch.

### 3.3.4.3 Neighborhood Intersections

There is a minor but important distinction between using sketches and using an adjacency matrix to calculate the weight of a neighborhood intersection. With sketches, \( d \) estimates are calculated for the intersection weight, where the \( i \)th estimate is the sum of the cells that are nonzero in the \( i \)th row of both sketches. Then, the estimated weight of the neighborhood intersection is the minimum of the \( d \) estimates. In addition to minimizing the individual edge weight errors, this minimizes the weight of false positives that occur when vertices \( u \in N(u) \setminus N(v) \) and \( v \in N(v) \setminus N(u) \) hash to the same cell in the sketch.

### 3.3.5 Total Score

As the total score is a linear combination of the three individual scores, the error is a linear combination of the error of each score. The bound below follows.

**Theorem 3.1.** Using a CM sketch of width \( w = \lceil \frac{e}{\varepsilon_1} \rceil \) and depth \( d = \lceil \ln \frac{1}{\delta} \rceil \) for estimating sample scores, a CM sketch of \( w = \lceil \frac{e}{\varepsilon_2} \rceil \) and \( d = \lceil \ln \frac{1}{\delta} \rceil \) for estimating preferential attachment scores, and \( |V| \) CM sketches of \( w = \lceil \frac{e}{\varepsilon_3} \rceil \) and \( d = \lceil \ln \frac{1}{\delta} \rceil \) for estimating homophily scores, the total estimated score of an edge \((u,v)\), \( s_{uv}^e \), is bound by the following inequality with probability at least \( 1 - \delta \).

\[
s_{uv} \leq s_{uv}^e \leq s_{uv} + \alpha \varepsilon_1 + \beta \varepsilon_2 + \gamma \varepsilon_3 N_{uv} \quad (3.7)
\]

where \( \alpha, \beta, \) and \( \gamma \) are weighting coefficients such that \( \alpha + \beta + \gamma = 1 \).

**Proof.** It follows from the linearity of expected values and Lemmas 3.1, 3.2, and 3.3.

The bound above holds only when each vertex uses their own neighborhood sketch. When employing the shared sketch heuristic, the homophily score, and thus the total score, is not bound. However, we show in Section 3.5 that the heuristic performs very well in practice.

### 3.4 Complexity Analysis

Let \( w_1 \) (\( w_2 \)) be the width of the sketches used for storing vertex (edge) weights, \( w_3 \) be the width of the sketches storing vertex neighborhood information, \( d \) be the depth of all the sketches, and \( l \) be the number of sketches used in the shared sketch heuristic \((l = |V|)\) in the individual
Each edge in the stream is processed in $O(w_3 d)$ time, and the total space required is $O(w_1 d + w_2 d + w_3 d l)$.

### 3.4.1 Time

The run time is determined by the operations required to update the sketches and score an edge.

**Model Updates.** When a new edge is added to the model, all of the corresponding weights must be updated. To update the weight of an edge (vertex), all of the $d$ cells in the sketch corresponding the edge (vertex) must be incremented, requiring $d$ hashes, and thus $O(d)$ time. Updating the neighborhood information for the two vertices incident on the edge requires $2d$ hashes, plus an additional two for the shared sketch heuristic, and is thus $O(d)$. Model updates are therefore $O(d)$.

**Edge scoring.** The sample (preferential attachment) score requires one (two) normalized weight estimations. Similar to updating the weights, querying a weight requires $O(d)$ time, hence the sample and preferential attachment scoring functions are $O(d)$. Calculating the homophily score of an edge $(u,v)$ requires traversing two sketches, $O(w_3 d)$, and normalizing by two weight estimations, $O(d)$. Therefore, scoring an edge is $O(w d)$.

**Total.** Processing each edge in the stream then requires $O(d + w_3 d) = O(w_3 d)$ time.

### 3.4.2 Space

The required space is proportional to the size of the sketches used, and the number of sketches used. The vertex and edge sketches require $O(w_1 d)$ and $O(w_2 d)$ space, respectively. Approximating neighborhood information requires $l$ sketches, and therefore $O(w_3 d l)$ space. In total, $O(w_1 d + w_2 d + w_3 d l)$ space is required.

### 3.5 Experiments

For our experiments, we have implemented the exact, approximate, and heuristic versions. As a baseline, we implemented the graph stream outlier detection approach proposed in [5]; see Section 3.6 for more details of this approach. Similar to our edge score, each edge in the baseline is assigned a probability of appearing.

#### 3.5.1 Experimental Setup

All experiments were performed on a dedicated Intel server consisting of two hex-core E5645 processors and 64GB DDR2 RAM. The data was stored on a 2TB RAID1 partition, and the operating system was installed on a 120.5GB SSD. The approximate and heuristic versions are written in C++, and the exact version is written in MATLAB. An exact version was also written.
in C++, however, it ran significantly slower than the equivalent version in MATLAB due to the handling of sparse matrix operations, so the MATLAB code was used for all experiments. The C++ code was compiled using GCC 4.8.2 with no optimization flags.

3.5.2 Dataset Descriptions

3.5.2.1 Enron

The original Enron email dataset was downloaded from https://www.cs.cmu.edu/~./enron/ and processed into an edgelist. All emails associated with the Enron domain that were in the To, From, CC, or BCC fields of the emails were considered for creating edges. One email generated an edge for each unique (sender, receiver) pair. The edge list is chronologically ordered, but for a single email the ordering of the (sender, receiver) pairs generated is random.

In total, six types of anomalous edge lists were created. Three had outliers injected throughout the entire stream, while the other three had outliers only in the second half of the stream. Each of these two categories contained the same three types of injections: (1) 1% outliers. For every edge in the stream, there was a 1% chance to randomly inject an outlier (i.e., anomalous edge); (2) 5% outliers. For every edge in the stream, there was a 5% chance to randomly inject an outlier; (3) 1% bursty outliers. For every edge in the stream, there is a .1% chance to inject a group of 10 outliers.

3.5.2.2 DBLP

DBLP contains records of publications from the computer science domain, containing the title of the publication, authors, year of publication, and more. The original data was downloaded from http://dblp.uni-trier.de/xml/ and processed into an edgelist of co-author pairs. All conference papers, journal articles, and books from 1954 to June 10, 2015 were used. We process the data and create a timestamped author pair edge list stream by generating an edge for every author pair on every paper. For example, if a paper published had 3 authors, then \( \binom{3}{2} \) edges are generated. As only the year of publication is available, the stream is processed chronologically and then alphabetically within each year.

3.5.2.3 IMDB

IMDB contains information about movies and television shows from the past few decades. The original data was downloaded from http://www.imdb.com/interfaces and processed into an edgelist where actors are vertices and an edge connects two actors if they co-star in a movie or TV show. All productions from 1970 until 2006 were used.
3.5.3 Scalability

To test the scalability of our proposed method, we generated several synthetic datasets using the Random Typing Generator (RTG) [8]. RTG is the state-of-the-art method for generating dynamic networks that follow several important properties present in real-world networks, e.g., power law degree distribution. As such, it enables us to generate realistic datasets for scalability testing.

![Scalability Testing: Run Time vs. Stream Size](image)

**Figure 3.3** Run times for various approximation parameters compared to the exact implementation. For all approximations, the vertex and edge weight sketches had $\varepsilon = 10^{-6}$ and $\delta = 10^{-3}$. The shared sketch heuristic runs both had a length of $l = \lceil \frac{\varepsilon}{0.001} \rceil$.

As shown in Figure 3.3, the run time increases linearly with the size of the stream, regardless of the chosen parameter values. However, the effect the sketch parameter values have on the size of the constant is evident. As is typical with approximation algorithms, there is a trade-off between accuracy and efficiency. As the error bounds shrink ($\varepsilon$ shrinks) or the probability of the
Table 3.2 ROC AUC comparisons for vertex and edge weight sketch parameters $\epsilon = .000001$, with the length of the shared sketch size shown above, all sketches set $\delta = .001$

<table>
<thead>
<tr>
<th>Injection Type</th>
<th>Exact $w = N/A$</th>
<th>Baseline $l = N/A$</th>
<th>Individual sketches $w = \lceil \frac{\epsilon}{.001} \rceil$</th>
<th>Shared sketches $l = \lceil \frac{\epsilon}{.001} \rceil$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full: bursty</td>
<td>0.971951</td>
<td>0.849379</td>
<td>0.974366</td>
<td>0.954515</td>
</tr>
<tr>
<td>Full: 1%</td>
<td>0.972161</td>
<td>0.848936</td>
<td>0.974553</td>
<td>0.954966</td>
</tr>
<tr>
<td>Full: 5%</td>
<td>0.973332</td>
<td>0.864021</td>
<td>0.974064</td>
<td>0.952195</td>
</tr>
<tr>
<td>Half: bursty</td>
<td>0.967197</td>
<td>0.844626</td>
<td>0.968344</td>
<td>0.942415</td>
</tr>
<tr>
<td>Half: 1%</td>
<td>0.967260</td>
<td>0.844871</td>
<td>0.968444</td>
<td>0.941580</td>
</tr>
<tr>
<td>Half: 5%</td>
<td>0.967735</td>
<td>0.844603</td>
<td>0.968337</td>
<td>0.939952</td>
</tr>
</tbody>
</table>

bounds holding increase ($\delta$ decreases), the run time increases commensurately. The dominating run time factor for the approximate and heuristic versions is the width of the sketches used for calculating the homophily score. When using equal widths for their neighborhood sketches, the approximate and heuristic versions have similar run times, with the heuristic taking slightly longer due to the higher number of nonzero cells in the shared sketches.

In all cases, the exact version took orders of magnitude longer to process the stream. Given the memory constraints of the exact version, it was necessary to store the graph using a sparse matrix representation. As a result, all update operations require significantly more time compared to using a dense matrix, contributing to the slow run time observed. For example, updating the weight of an edge using a dense matrix is $O(1)$, while it is $O(|V|)$ in a sparse matrix (though, typically much better in practice due to network sparsity).

The baseline run time is omitted as our implementation may be less efficient than that of the authors, with ours taking multiple days per edge stream. As the baseline is designed for batch updates, where each new graph in the stream triggers an update, processing edge streams (where each edge is treated as its own graph) is expected to be a worst-case scenario. Each new edge requires multiple edge-set samples to update their spanning forests and the statistics for each connected component.

### 3.5.4 Identifying Injected Outliers

As no ground-truth data is available for testing the precision of our methods, we must manually create a ground-truth dataset. We follow a method outlined in [10] and inject outliers into the well-studied, real-world Enron email dataset. Artificial outliers are injected by sampling two vertices from a uniform distribution and adding an edge between them. This generation opposes the preferential attachment model, where vertices are connected with probabilities proportional to their degrees and, as such, contradicts the shown power law distribution of the vertex degrees.
and edge multiplicities [57].

In total, six types of anomalous datasets were created by varying the percent of outlier edges in the stream, as well as their location (for full details see Section 3.5.2). As the injection process was random, 10 edge streams were created for each dataset type, creating a total of 60 anomalous edge streams. All of the results shown for each dataset type are averages taken over the 10 edge lists.

### 3.5.4.1 Approximation Effects on Precision

We measure the precision of our method by comparing the area under the curve (AUC) of the receiver operating characteristics (ROC) curve for various sketch parameter values. The ROC curve is formed by plotting the true positive rate (TPR) against the false positive rate (FPR) for a range of outlier threshold values. For calculations, we consider the positive class the outlier edges we manually injected, and the rest of the stream the negative class. The TPR is proportional to the number of edges in the positive class that were correctly classified (labeled) as outliers, and the FPR is proportional to the number of edges in the negative class that were falsely classified as positives (i.e. normal edges being classified as outlier edges). The AUC is thus in the range $[0, 1]$, where 1 is a perfect classifier. Table 3.2 shows the results for the exact, approximate, heuristic, and baseline implementations.

**Table 3.3** Approximation error as a result of using sketches. The values shown for IMDB and DBLP are median $\times 10^6$ ($\pm$std $\times 10^3$), for the Enron datasets (Full and Half datasets) the values shown are median $\times 10^6$ ($\pm$std $\times 10^9$). For all runs, the vertex and edge weight sketches had $\varepsilon = 10^{-6}$ and $\delta = 10^{-3}$. Median and standard deviation taken over 10 runs.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$s^t$</th>
<th>$s^b$</th>
<th>$s$</th>
<th>$s^t$</th>
<th>$s^b$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMDB</td>
<td>0.07 ($\pm$0.53)</td>
<td>0.07 ($\pm$0.53)</td>
<td>0.00 ($\pm$0.09)</td>
<td>0.07 ($\pm$0.53)</td>
<td>0.07 ($\pm$0.53)</td>
<td>0.00 ($\pm$0.09)</td>
</tr>
<tr>
<td>DBLP</td>
<td>0.00 ($\pm$0.23)</td>
<td>0.00 ($\pm$0.23)</td>
<td>0.00 ($\pm$0.00)</td>
<td>0.00 ($\pm$0.23)</td>
<td>0.00 ($\pm$0.23)</td>
<td>0.00 ($\pm$0.00)</td>
</tr>
<tr>
<td>Full: bursty</td>
<td>12.69 ($\pm$13.64)</td>
<td>12.69 ($\pm$13.64)</td>
<td>0.00 ($\pm$0.00)</td>
<td>12.69 ($\pm$13.64)</td>
<td>12.69 ($\pm$13.64)</td>
<td>0.00 ($\pm$0.00)</td>
</tr>
<tr>
<td>Full: 1%</td>
<td>12.69 ($\pm$13.64)</td>
<td>12.69 ($\pm$13.64)</td>
<td>0.00 ($\pm$0.00)</td>
<td>12.69 ($\pm$13.64)</td>
<td>12.69 ($\pm$13.64)</td>
<td>0.00 ($\pm$0.00)</td>
</tr>
<tr>
<td>Full: 5%</td>
<td>10.67 ($\pm$7.91)</td>
<td>10.67 ($\pm$7.91)</td>
<td>0.00 ($\pm$0.00)</td>
<td>10.67 ($\pm$7.91)</td>
<td>10.67 ($\pm$7.91)</td>
<td>0.00 ($\pm$0.00)</td>
</tr>
<tr>
<td>Half: bursty</td>
<td>12.99 ($\pm$9.96)</td>
<td>12.99 ($\pm$9.97)</td>
<td>0.00 ($\pm$0.00)</td>
<td>12.99 ($\pm$9.96)</td>
<td>12.99 ($\pm$9.97)</td>
<td>0.00 ($\pm$0.00)</td>
</tr>
<tr>
<td>Half: 1%</td>
<td>12.99 ($\pm$9.96)</td>
<td>12.99 ($\pm$9.97)</td>
<td>0.00 ($\pm$0.00)</td>
<td>12.99 ($\pm$9.96)</td>
<td>12.99 ($\pm$9.97)</td>
<td>0.00 ($\pm$0.00)</td>
</tr>
<tr>
<td>Half: 5%</td>
<td>12.06 ($\pm$6.28)</td>
<td>12.06 ($\pm$6.29)</td>
<td>0.00 ($\pm$0.00)</td>
<td>12.06 ($\pm$6.28)</td>
<td>12.06 ($\pm$6.29)</td>
<td>0.00 ($\pm$0.00)</td>
</tr>
</tbody>
</table>

We first look at the performance of the individual sketch approximation compared to exact. Both achieved very high AUCs, around .97 for all datasets, indicating they were highly successful in identifying the outlier edges. Interestingly, with $w = \lceil \frac{\varepsilon}{\delta} \rceil$, the approximation yielded a higher AUC for all datasets. For this to happen, the error resulting from the sketches must have worked in favor of the classifier. Using a smaller width of $w = \lceil \frac{\varepsilon}{3\delta} \rceil$ we do not see this effect, and the AUC is slightly less than the exact version at around .966.
Next, we compare the shared sketch heuristic to the individual sketch approximation. For equal widths, the individual sketches perform better, as expected, due to the smaller error on the homophily scores. A key difference between the two is the effect of the sketch widths. When the individual sketches go from $\varepsilon = 10^{-3}$ to $10^{-2}$ there is very little change in the AUC. The same change in size causes a drop of around .05 for all datasets in the shared sketch heuristic. Again this is expected, as the number of collisions (i.e. error) due to sharing is proportional to the width of the sketch.

Finally, we compare our implementations to the baseline. We modified the baseline method very slightly in order to improve the accuracy: Instead of placing every component smaller than $\text{MIN\_COMPONENT\_SIZE}$ into a single outlier component, we left them in their own component. When they were grouped into a single, large outlier component the AUC was near zero, as the outlier edges were connecting vertices in this component, resulting in a very high non-sample score for the edge. With our modification, the AUC for the baseline rose an order of magnitude from .08 to .84. However, both the approximate and heuristic versions of our method outperform the baseline on all datasets. We are able to achieve a higher AUC than the baseline in addition to providing constant space and time complexities (Section 3.4). No time or space complexity analysis for the baseline is shown in [5].

### 3.5.4.2 Approximation Effects on Scoring

Using a CM sketch to store the edge stream model introduces approximation error to the scores calculated for every edge in the stream. Table 3.3 shows the error caused by approximation on a variety of datasets.

We focus on the error of the homophily score, as that is the most volatile and parameter sensitive. For all of the Enron datasets, the homophily score has a median value of 0, with a standard deviation of 0 (to the ninth decimal place at least). However, scores for IMDB and DBLP show a very different result; while medians are still 0, standard deviations are on the order of at least $10^{-4}$. This large discrepancy indicates that there are a few very large hub vertices in the network, causing a high number of collisions in the neighborhood sketches. Most of the vertices have a very accurate picture of their neighborhood, having only a few neighbors, hence the median error being low. Further evidence of this phenomenon is the low error of the individual sketches compared to the shared sketches. No vertex shares a neighborhood sketch with a hub vertex, thus the error is confined to only the edges incident to hub vertices.

### 3.5.5 Real-World Outliers

While synthetic datasets are ideal for unit testing components of an algorithm and providing ground truth, it is also illustrative and insightful to examine real-world use cases and outliers.
Here we highlight a few examples of real-world outliers found using our approach.

### 3.5.5.1 DBLP

DBLP contains records of publications from the computer science domain, containing the title of the publication, authors, year of publication, and more. For details of this dataset, and the IMDB dataset, see Section 3.5.2. We perform a few case studies to highlight real world outliers found in the data.

**Case Study 1.** Pedro Olmo S. Vaz de Melo, Christos Faloutsos, and Antonio Alfredo Ferreira Loureiro. *Human Dynamics in Large Communication Networks.* SDM. 2011.

This is the first paper that we will look at that generated outliers (anomalous co-author pairs). This paper illustrates two interesting circumstances that can cause outliers, the bridging of fields of study and the overcoming of geographic boundaries. Christos Faloutsos primarily publishes in the fields of data mining and knowledge discovery, while, at the time of this publication, the first and third author primarily worked in the fields of distributed computing and wireless sensor networks. Moreover, Pedro Melo and Antonio Loureiro are researchers at universities in Brazil, while Christos Faloutsos is a professor at Carnegie Mellon in Philadelphia. So, how did this collaboration come to be? The lead author, Pedro Melo, was a visiting researcher for one year at Carnegie Mellon during his PhD. This paper is one of the first in what has become a regular collaboration among the authors, as shown by their numerous joint publications since this paper. Our algorithm successfully identified this new collaboration between Pedro and Christos and labeled it as an outlier.

**Case Study 2.** Charu C. Aggarwal, Yuchen Zhao, and Philip S. Yu. *A framework for clustering massive graph streams.* Statistical Analysis and Data Mining: The ASA Data Science Journal 3.6 (2010): 399-416.

Unlike Case Study 1, all of the authors from this paper live in the United States and have similar areas of research. This paper highlights a third cause of outliers; namely, the beginning of an author’s career and collaborations. Prior to this paper, Charu Aggarwal and Philip Yu had a number of publications, including a co-authored book. At the time of this publication, however, Yuchen Zhao was a new PhD student at UIUC under Philip Yu. This was the first publication showing the collaboration of the authors. Our algorithm correctly labeled the edge between Charu and Yuchen as an outlier and elucidated the beginning of Yuchen’s publishing career and the beginning of his collaboration with Charu.

---

*However, another paper was published the same year with the same authors, but due to the titles of the papers and the alphabetical ordering of the stream within years, the earlier work was placed downstream from this paper.*
3.6 Related Work

Our method complements the existing body of work, evaluated extensively in Chapter 2, in a number of ways. We focus on the contrasts with [5], as it is the closest related work. In [5], Aggarwal et. al. propose a structural reservoir sampling algorithm that heuristically retains a subset of all edges seen in the graph stream. When a new graph arrives, the probability of each edge is calculated using the edges in the reservoir sample, and then aggregated into a single graph likelihood value. Graphs with a likelihood below a threshold value are output as outliers. Sampling enables exact calculations to be performed, but relies on salient portions of the stream to be retained, with all other information being lost. Conversely, we maintain an approximate model of the entire edge stream and thus use approximate calculations in our analysis. Using a sketch-based approximation enables a well-defined (constant) upper bound on the space used, and we are able to prove probabilistic error bounds on the scoring functions used. Moreover, model updates and edge scoring are performed in constant time. Graph snapshot analyses typically perform more expensive operations, such as dynamically maintaining spanning forests [5], relying on the updates being less frequent, and thus are not well-suited for edge streams.

3.7 Conclusion

Outlier detection in dynamic networks has been a topic of growing interest in the past few years. While it has traditionally focused on graph streams, in this chapter, we present the first approach for outlier detection in edge streams. Our sketch-based approach provides constant time updates and scoring functions, in addition to requiring constant space that is independent of the size of the edge stream and network. Experiments on synthetic datasets have shown that our approximations have only a marginal impact on the accuracy of the scoring functions, and we are still able to achieve high precision on identifying injected outliers, outperforming the baseline method. Mining the DBLP co-authorship network elucidates some interesting types of outliers that exist, and our ability to capture them. In short, we have presented a scalable and effective method for outlier detection in edge streams.
CHAPTER 4

ANOMALIES IN HYPERGRAPH EDGE STREAMS

4.1 Introduction

Restricting the graph to pairwise edges, implying all relationships or interactions in the system occur in pairwise manners, often results in a loss of information or unfaithful representation. Higher order relationships, among three or more entities, are in fact quite common. For example, consider the DBLP co-author network used in Section 3.5.2, where authors are vertices and edges connect co-authors of papers. Given the triangle of authors \((A, B, C)\), using pairwise edges it is impossible to know if the trio co-authored a single paper, or if three different papers were published by the pairs \((A, B), (A, C), (B, C)\). In this chapter we propose an outlier detection algorithm for streams of hyperedges, which generalize edges and connect any positive number of vertices. Similar to Chapter 3, two main research questions arise. First, how can we design an effective algorithm for identifying outliers in a hyperedge stream? Second, how can we make the algorithm time and space efficient, i.e. suitable for real-time analysis?

We propose, to the best of our knowledge, the first method for identifying outliers in a hyperedge stream. The major contributions of this chapter can be summarized as follows.

1. **First Streaming Model for Outlier Detection.** We describe a data driven model for scoring hyperedges in a continuous stream. Our model is based on the principle of stream
self-similarity and co-occurrence of vertices. (Section 4.3)

2. **Scalable Approximation Model.** To effectively employ our model in streaming environments, we propose an approximation based on MinHash and Locality-sensitive Hashing (LSH). (Section 4.4)

3. **Extensive Experiments.** Extensive experiments are performed to test scalability and empirically test the effects of approximation on identifying injected ground-truth outliers. (Section 4.6)

### 4.2 Related Work

Outlier detection in hypergraphs, as well as bipartite graphs, is a relatively unexplored area of research. While we consider only hypergraphs, it is necessary to also consider any method proposed for bipartite graphs, given the relationship between the two.

One of the earliest approaches to leverage a hypergraph model was by Wei et al. [180]. Given a static database, where objects have categorical attributes, a hypergraph is constructed by making row objects the vertices, and frequent itemset patterns the hyperedges. If a row matched a given itemset pattern, its representative vertex was part of that itemset pattern’s hyperedge. In this way, there is an assumption of homogeneity of vertices within a hyperedge—as all the rows that match the same itemset pattern are expected to be similar. Outliers are then (vertex, hyperedge) pairs, where the vertex deviates significantly from the other vertices in the hyperedge on some attribute. A similar assumption is made in [165], where a query-driven technique for outlier detection in bipartite graphs is proposed. Given the bipartite graph $G = (V_1 \cup V_2, E)$ and a query vertex $v \in V_1$, outliers are vertices in $V_2$ that are adjacent to $v$, but do not share many connections with the other neighbors of $v$. As one partition of the graph represents the hyperedges, outliers are again defined by a (vertex, hyperedge) pair. Unlike these approaches, we make no assumption of homogeneity, as a hyperedge could contain a set of vertices that would be disconnected if not for itself. In fact, the degree to which a hyperedge is an outlier in our model is defined by this. If a single vertex were dissimilar from the rest (as is the case for the two papers mentioned), the outlier score would be considerably lower than if all the vertices were dissimilar. Moreover, we consider a dynamic data model, specifically, a streaming data model, as opposed to static.

A complementary approach to the above is to assume that vertices in the graph participate in hyperedges with a fixed probability [157–159]. Hyperedges are then a sampling of the vertices, where the likelihood of a hyperedge is inversely proportional to its outlieriness. The hyperedge set of a graph is then assumed to be drawn from a mixture model, where the majority of the hyperedges come from the true, nominal, vertex participation distribution, and a small fraction
are drawn from an anomalous vertex participation distribution (e.g. a uniform distribution). Implicit in this model is an assumption of independence of the vertices—meaning if a hyperedge is known to contain a given vertex, that does not affect the probability of it containing any other vertex. This is the key difference between their mixture model approach and ours. We explicitly incorporate the pairwise and higher order relationships of vertices into the calculated outlier scores. Moreover, this method is also designed for static hypergraphs, with no obvious extension to dynamic hypergraphs or hyperedge streams.

A dynamic hypergraph model is used by Park et al. [132] in their work with the Enron graph. They extend their previous work with scan statistics on directed graph streams [136] to hypergraph streams. Each object in the stream is a hypergraph, representing the activity during a window of time. Thus, outliers are entire hypergraph objects. We take a more fine-grained approach and consider a stream of hyperedges, annotating each incoming edge with an outlier score.

In short, we propose the first method for outlier detection in hyperedge streams. We make no assumptions about the homogeneity of the graph, or the hyperedges themselves, and annotate every hyperedge in the stream with an outlier score.

### 4.3 Problem Statement and Model

Streaming algorithms are commonly developed to analyze a static dataset that is too large to analyze all at once. In such cases, it is common to impose a random or even adversarial ordering of the data in the stream, to provide a worst case analysis. Unlike these methods, we assume our stream is being produced by a dynamic generating process, such that the ordering of the stream is defined implicitly by when each hyperedge is generated. For example, if we are analyzing a stream of emails, then each hyperedge represents the list of recipients, and its place in the stream is dictated by the time it is sent.

We informally define the problem we address as follows.

**Problem 1. (Outlier detection in hyperedge streams)**

**Given:** A continuous hyperedge stream $\mathcal{E} = \{(u, v, w, \ldots), \ldots\}$, where each tuple is a hyperedge.

**Find:** Hyperedges in the stream that deviate from the expected structural patterns.

It is worth repeating the clarification of Chapter 3. There is a subtle distinction between our problem statement and that of anomalies in link prediction. Here we are identifying outliers based on hyperedges that do appear in the stream, compared to link prediction anomalies which are hyperedges that do not appear but are expected to.
4.3.1 Data Model

Let $V = \{v_i\}$ be a finite vertex set. We do not assume the complete vertex set is known \textit{a priori}, allowing new vertices to be added over time. However, we do assume that every vertex is assigned a unique label or id, $l(v)$, meaning $l(v_i) = l(v_j)$ if and only if $v_i = v_j$. We define a hyperedge stream $\mathcal{E} = \langle he_1 = (u, v, w), he_2, \ldots \rangle$ as the continuous sequential arrival of hyperedges, where each hyperedge is a subset of vertices with a minimum cardinality of two. Hyperedges are undirected, may appear in the stream an arbitrary number of times, and are considered insertions. As such, the underlying hypergraph, if we were to compose a single static hypergraph from every hyperedge received, could have parallel hyperedges. The size of a hyperedge, $|he_i|$, is the number of vertices it contains.

4.3.2 Hyperedge Scoring and Outlier Detection

Loosely, outliers are hyperedges which deviate significantly from the expected structural behavior of the hypergraph. Of the numerous ways to define expected behavior, we choose a self-similarity (frequency based) data driven approach. The key idea is to use historic data from the stream to determine the “normality” of the next incoming hyperedge. Normality will be defined by the latest hyperedge’s similarity to all previously seen hyperedges in the stream.

This definition of normality was chosen for two reasons. First, it is well known that the more an object has been seen in a stream, the more likely it is to be seen again. Second, we experimentally show in Table 4.1 that vertices do not appear in the stream independently of each other. These dependences result in frequently co-occurring pairs or higher order sets of vertices, but not necessarily the exact same sets each time.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|}
\hline
Dataset & Percent Dependent \\
\hline
Enron & 89.43 \\
DBLP & 99.94 \\
Movielens & 74.84 \\
IMDB & 99.62 \\
\hline
\end{tabular}
\caption{Results of vertex independence tests for neighbors in each dataset. To test if two vertices were independent of each other, we ran Fisher’s exact test on their hyperedge incidence vectors and tested at a significance level of 0.05. Each dataset had 10,000 pairs of neighbors randomly sampled and tested.}
\end{table}

A straightforward approach for scoring the normality of the newest hyperedge is to find the frequency of that hyperedge in the history of the stream. Taking the outlier score as the
complement of the frequency, we get the following scoring function,

$$o(he_i) = 1 - \frac{\text{count}(he_i)}{m}$$

(4.1)

where $m$ is the size of the stream\(^1\). However, this approach, using hard matching (HM), is far too constrained and prone to false positives. Figure 4.1 illustrates this.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{hard_matching.png}
\caption{Example of scoring hyperedges in a stream using hard matching. Hard matching is too constrained for obtaining meaningful results. Even though vertices \{1, 3\} co-occur in $he_1$, $he_2$ is given an outlier score of 1. Similarly, $he_3$ is given a score of 1 even though \{2, 3\} co-occur in $he_1$.}
\end{figure}

To capture these pairwise and higher order similarities, we propose using a soft count, where we relax the requirement of the hyperedges exactly matching. Instead of a binary 0/1 matching function, we will use a soft matching function that outputs a real number in the range \([0,1]\). In soft counting, the number of hyperedges above a user-defined soft matching threshold is the final count. We will use the Jaccard similarity for our soft matching function. Figure 4.2 shows how switching from hard to soft matching changes the outlier scores from Figure 4.1.

Figure 4.3 illustrates the drastic effects this switch from hard to soft matching has. The top two figures show the number of times each unique hyperedge is repeated in the real-world streams for Enron and IMDB\(^2\). Hyperedge repetition in each stream seems to follow a powerlaw distribution, with the vast majority of hyperedges appearing only once or twice. However, when we switch to soft matching, the high degree of similarity among the hyperedges in the stream can be seen. Figures 4.3c and 4.3d show substantial shifts in the distribution for hyperedge repetition. As the threshold for a match decreases, the number of matches each unique hyperedge has in the stream increases.

---

\(^1\)To handle the first object in the stream we take the score to be 1. Another approach is to use $m + 1$ in the denominator, which as the stream grows is approximately equal to using $m$.

\(^2\)Dataset descriptions are provided in Section 4.6.1
Using the proposed soft count, an incoming hyperedge is scored as follows.

\[ o(he_i) = 1 - \frac{\text{count}_\text{soft}(he_i)}{m} \]  (4.2)

Again we have mapped the problem of outlier detection from the graph domain to the time series domain, for which a number of well established techniques exist [84]. In this work, we choose a simple thresholding approach, but any approach that can evaluate a new data point in constant time is suitable. We can now restate Problem 1 more formally. Given a continuous hyperedge stream \( E \), and a soft matching threshold \( \tau \), identify all outlier edges \( he \in E \) such that \( o(he) \geq \tau \).

### 4.4 Model Approximation

Explicit and exact storage of the network structure for on or offline analysis is infeasible for streams at even a moderate scale. Instead, we propose using MinHash and locality sensitive hashing (LSH) to store the salient structural information from the stream, enabling an efficient approximation of our soft counting function. We first give a brief overview of MinHash and LSH (for more details see [30, 102]), then describe how it fits into our proposed approach.

#### 4.4.1 MinHash

MinHash is a scheme devised by Broder [31, 32] for efficiently estimating the similarity of two sets of items. Specifically, it estimates the Jaccard Similarity of two sets, which is one of the most common and effective similarity metrics. The Jaccard Similarity between two sets \( X \) and \( Y \) is defined as \( J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|} \). If the sets are large then this calculation is inefficient. Instead, MinHash computes a sketch for each set, where the size of the sketch is much smaller than the
Figure 4.3 How many times each hyperedge is repeated, i.e. it’s count, in the Enron dataset. Figure 4.3a shows hard count, Figure 4.3c shows soft count with $\tau = 0.25$.

The sketch of a set, called the MinHash signature of the set, relies on computing $k$ minhash values, each using a different hash function. Let $h$ be a hash function that maps elements of any set $S$ to distinct integer values. The minhash of $S$, $h_{\text{min}}(S)$, is defined as the minimum hash value $h(x)$ over all $x \in S$. The MinHash signature of $S$ is the concatenation of $k$ minhash values.

It is fairly simple to show that for two random sets $X$ and $Y$, $P[h_{\text{min}}(X) = h_{\text{min}}(Y)] = J(X,Y)$. Using this result, $J(X,Y)$ can be estimated by comparing the length $k$ MinHash signatures of $X$ and $Y$. Let $w$ be the number of places the signatures have the same value, i.e.

---

3We use the uppercase MinHash when referring to the scheme, and the lowercase minhash when referring to a computed value.
Figure 4.4 Example of computing a single MinHash value for a set. The MinHash value is the minimum value of the set with respect to the chosen hash function \( h \).

\[
h_{\text{min}}(S) = \arg \min_{x \in S} h(x)
\]

\[
S = \begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\quad \rightarrow \quad h(1) = 12 \\
\quad \rightarrow \quad h(2) = 47 \\
\quad \rightarrow \quad h(3) = 5 \\
\rightarrow \quad h_{\text{min}}(S) = 3
\]

the number of times \( h_{\text{min}}(X) = h_{\text{min}}(Y) \), then the ratio \( w/k \) is an unbiased estimate of \( J(X,Y) \).

4.4.2 LSH

The goal of locality sensitive hashing (LSH) is to map similar items into similar “buckets” with high probability, where the number of buckets is much smaller than the number of items. Using the MinHash signatures of the sets, an efficient LSH approach can be developed for indexing the sets in a way such that pairs of sets above a given similarity threshold are likely stored in similar buckets.

The LSH index is stored using \( b \) hash tables. Each set will be mapped into one bucket in each of the hash tables, where the ID of the set will be stored, based on a portion of its MinHash signature. For a given set, the MinHash signature will be broken into \( b \) bands of length \( r \). Each band corresponds to one of the hash tables, and the \( r \) length portion of the signature for that band is hashed to determine the bucket the set ID is placed in. Using the fact that similar sets have similar signatures, and thus are likely to fall into the same bucket in a least one of the hash tables, we can query the index with a set \( Q \), and find all approximately similar sets by taking the union of the buckets that \( Q \) maps to.

Lemma 4.1. [30] Given a set \( Q \), the probability that a set \( R \) with a true Jaccard similarity of \( s \) falls into at least one of the same buckets as \( Q \), and is thus returned as a match, is \( 1 - (1 - s^r)^b \).

Proof. The probability that the signatures of \( Q \) and \( R \) agree in any particular location of their signatures is \( s \). Therefore, the probability they agree in all rows of a given band is \( s^r \), and the probability they disagree in at least one row of a given band is \( 1 - s^r \). Thus, the probability the signatures disagree in at least one row of every band is then \( (1 - s^r)^b \), and the probability the signatures agree in all rows of at least one band is \( 1 - (1 - s^r)^b \). \( \square \)
Figure 4.5 MinHash signatures are formed by $k$ minhash values, each generated by a different hash function. Signatures are then broken into $b$ bands of length $r$ for storing in the hash tables comprising LSH index. In this figure we show a single hash table for every band, but in reality each band has its own hash table.

4.4.3 Hyperedges Streams, MinHash, and LSH

A hyperedge is simply a set of vertices, and thus hyperedges can be compared directly using the MinHash scheme discussed in Section 4.4.1. Vertices constitute the items of the set, and the vertex labels are hashed to create the hyperedge’s MinHash signature. As each hyperedge is received in the stream, it is scored according to Eqn. 4.2, where the count is taken from the LSH index. Specifically, the count is the cardinality of the union of buckets that the hyperedge is hashed into. Once the hyperedge is scored, the LSH index is updated by placing the hyperedge ID into its appropriate buckets in each hash table. Figure 4.5 shows an example of this, where the $b$ hash tables are collapsed into a single one for illustrative purposes.

Estimating the count using LSH introduces false positives and negatives. Consequently, instead of getting the exact outlier score, we compute an estimated outlier score, $o_e(he)$. The following lemma shows the boundary expected values of the estimated outlier score.

**Lemma 4.2.** Consider a hyperedge $he_i$ with $m' < m$ historical hyperedges with a similarity less than the soft matching threshold $\tau$. Using MinHash signatures with $b$ bands of length $r$, the expected score of an edge is bound by the following inequality

$$o(he_i) - \frac{(m - m')(1 - \tau)^b}{m} \leq \mathbb{E}[o_e(he_i)] \leq o(he_i) + \frac{m'(1 - (1 - \tau)^b)}{m}$$

**Proof.** The exact outlier score is $o(he_i) = 1 - \frac{\text{count}_{(he_i)}}{m}$. Using MinHash and LSH, we are approximating the soft count value for the hyperedge. Our estimated scoring function is then $o_e(he_i) = 1 - \frac{\text{count}_e(he_i) + fp - fn}{m}$, where $fp$ and $fn$ are the false positives and false negatives,
respectively. By the linearity of expected values,

\[ E[o(he_i)] = E \left[ 1 - \frac{\text{count}_s(he_i) + fp - fn}{m} \right] \]

\[ = 1 - \frac{1}{m} \left( \text{count}_s(he_i) + E[fp] - E[fn] \right) \]  

(4.3)  

(4.4)

From Lemma 4.1 we know the probability of a hyperedge \( he_j \) with similarity \( s \) matching with \( he_i \) is \( 1 - (1 - s^r)^b \). For hyperedges with \( s < \tau \), a match indicates a false positive. For hyperedges with \( s \geq \tau \), not matching indicates a false negative. Hence \( p(fp) = 1 - (1 - s^r)^b \) for all \( he_j \) such that \( J(he_i, he_j) = s < \tau \) and \( p(fn) = 1 - (1 - (1 - s^r)^b) \) for all \( he_j \) such that \( J(he_i, he_j) = s \geq \tau \).

**Upper bound.** The upper bound assumes no false negatives, as they would counteract the false positives, and the worst case expected value for the false positives. Define variables \( X_j \) as indicator variables, where \( X_j = 1 \) if \( J(he_i, he_j) < \tau \) but \( he_j \) is reported as a match by the LSH index, 0 otherwise. Let \( Z = \sum_{j=1}^{m'} X_j \) be a random variable for the number of false positives returned by the LSH index. An upper bound on the expected number of false positives can be obtained by assuming all \( he_j \) have a similarity some small epsilon below \( \tau \). Therefore, \( E[Z] \leq E[\sum_{j=1}^{m'} X_j] \leq m' \left( 1 - (1 - \tau^r)^b \right) \).

**Lower bound.** The lower bound assumes no false positives and the worst case expected value for the false negatives. By an analogous argument to the expected number of false positives, the worst case expected number of false negatives is \( (m - m')(1 - \tau^r)^b \).

4.5 **Complexity Analysis**

Let \( k \) be the number of hash functions used for computing MinHash signatures, \( b \) be the number of bands used in the LSH scheme for signature hashing, \( B \) be the number of bins used in each hash table, \( m \) be the number of hyperedges processed by the stream, and \( |he| \) be the size of the current hyperedge.

4.5.1 **Time**

For an incoming hyperedge, the main operations are performed are: (1) computing it’s MinHash signature; (2) computing it’s LSH bins; (3) counting the number of matches; (4) outlier scoring. Each part is independent and thus they are additive in their complexities, so we consider each step in turn.

1. **Compute MinHash Signature.** Assume that \( k \) hash functions are being used. Each hash function is applied to every vertex in the hyperedge, resulting in \( \Theta(k|he|) \) operations.

2. **Compute LSH.** Assume a hash table is used to store the bins for LSH, up to a maximum
of $B$ bins. Each hyperedge is inserted into each of the $b$ hash tables, with a single hash
being computed for identifying each bin in each table, resulting in $\Theta(b)$ operations.

3. **Count Matches.** Each bin holds hyperedge labels which have a band that hashes to that
bin, thus we require the union of these sets. If we assume a uniform distribution of the
hyperedges over all the bins, the expected number of labels in each bin is $\lceil \frac{m}{B} \rceil$. In the
worst case, with hash based sets, taking the union will require $O(b\lceil \frac{m}{B} \rceil)$ operations.

4. **Calculate Outlier Score.** Once the count has been calculated, the outlier score just
requires computing Eq 4.2, which is $O(1)$.

In total, the worst case time complexity for annotating a hyperedge is $O(k|he| + b + b\lceil \frac{m}{B} \rceil + 1)$.
While the worst case complexity scales linearly with respect to the stream size, in practice we see
it is nearer to constant (see Figure 4.6). The key reason for this is that repetition of hyperedges
means we are not adding more items into the buckets in our LSH bins, as each bin is a set.

### 4.5.2 Space

The only scaling data structure required by our method is the hash tables for the LSH index.
Each hyperedge that is added to the LSH index stores $b$ IDs, thus the total space required is
$O(bm)$. While the memory scales with the size of the stream, only $b$ IDs are stored per hyperedge,
instead of having to store either the hyperedge itself, or the MinHash signature of the hyperedge.
If each label is a 4 byte integer, less than 4 GB of memory is required to process a stream with
1 billion hyperedges. However, as shown in Table 4.2 and Figure 4.7, when using Python and
string IDs the memory can grow quite rapidly.

### 4.6 Experiments

All methods were implemented using Python 2.7. The exact versions store a linked list of the
hyperedge stream, which is traversed each time a new hyperedge arrives. The approximate
version uses Python dictionaries (hash tables) to store the LSH index. Four datasets were used
in testing: DBLP, IMDB, Movielens, and Enron.

#### 4.6.1 Dataset Descriptions

We use four real-world datasets in our experiments, described below. In total, six types of
anomalous hyperedge lists were created from each dataset to test precision and recall. For details
about how outliers were injected, see Section 4.6.3. Three had outliers injected throughout the
entire stream, while the other three had outliers only in the second half of the stream. Each of
these two categories contained the same three types of injections: (1) 1% outliers. For every
hyperedge in the stream, there was a 1% chance to randomly inject an outlier (i.e., anomalous hyperedge); (2) 5% outliers. For every hyperedge in the stream, there was a 5% chance to randomly inject an outlier; (3) 1% bursty outliers. For every hyperedge in the stream, there is a .1% chance to inject a group of 10 outliers.

4.6.1.1 Enron

The original Enron email dataset was downloaded from https://www.cs.cmu.edu/~./enron/ and processed into a hyperedge list. All emails associated with the Enron domain that were in the From, CC, or BCC fields of the emails were considered for creating hyperedges. One email generated one hyperedge in the stream, and that hyperedge contained all emails in the above stated fields. The hyperedge list is chronologically ordered, simulating the original stream generating process.

4.6.1.2 DBLP

DBLP contains records of publications from the computer science domain, containing the title of the publication, authors, year of publication, and more. The original data was downloaded from http://dblp.uni-trier.de/xml/ and processed into a hyperedge list of co-author groups. Each paper in the list generated a single hyperedge in the stream, and contained all authors from the generating publication. All conference papers, journal articles, and books from 1954 to June 10, 2015 were used. As only the year of publication is available, the stream is processed chronologically and then alphabetically within each year.

4.6.1.3 IMDB

IMDB contains information about movies and television shows from the past few decades. The original data was downloaded from http://www.imdb.com/interfaces and processed into a hyperedge list, where actors are vertices and a hyperedge connects actors if they co-star in a movie or TV show. All productions from 1970 until 2006 were used.

4.6.1.4 Movielens

Movielens contains free text reviews and tagging activity submitted by users to the website movielens.org. The orginal data was downloaded from https://grouplens.org/datasets/movielens/ using the “latest” dataset, generated on January 29, 2016. Each user submitted review of a movie generates a hyperedge in our stream, where the free-text tags are the vertices incident on the hyperedge. Reviews from January 9, 1995 to January 29, 2016 were used.
4.6.2 Scalability

We look at the scalability of our methods from two angles: the average time required to process a hyperedge, the total time taken for the stream, and the memory usage of the model over time.

![Graphs showing average processing time for different datasets](image)

**Figure 4.6** Average time taken to process “batches” of 1,000 hyperedges. No model uses batch analysis, times were simply aggregated into batches for illustrative purposes. In DBLP the Exact versions were unable to complete in 24 hours, processing less than half the stream.

### 4.6.2.1 Average processing time

The average processing time per hyperedge is the inverse of the throughput of the method. Figures 4.6a and 4.6d show the average processing time for batches of 1,000 hyperedges in Enron and DBLP as the stream progresses. The first result that stands out is that the average time increases superlinearly for the exact methods, while our approximation appears to take near constant time, or linearly increasing time with a very low constant, per hyperedge for the duration
of the stream. Second, while the exact method produces smooth curves, our approximation has several points that are outliers, where certain batches took orders of magnitude longer to process than the preceding or future batches. This is likely a result of using hash tables for storing buckets. Certain buckets, due to hash collisions and high degree vertices, will fill much more than others. The time required to process a new hyperedge using MinHash signatures and LSH is proportional to the size of the buckets the new hyperedge is placed in, hence the increased processing time.

**Table 4.2** Average run time and peak memory usage over 10 runs using the real-world datasets. DNF indicates the method did not complete after 24 hours of run time. The run time for the exact version is an order of magnitude or more greater across all datasets, with DBLP not completing after 24 hours. Peak memory usage shows that there is a tradeoff, drastically reducing run time at the cost of memory. Within the approximation, using shorter MinHash signatures clearly has a large impact on both run time and memory.

<table>
<thead>
<tr>
<th>Method</th>
<th>Run Times (s)</th>
<th>Peak Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Enron Movielens IMDB DBLP</td>
<td>Enron Movielens IMDB DBLP</td>
</tr>
<tr>
<td>Exact $\tau = 1.0$</td>
<td>4016 6857 62906 DNF</td>
<td>1.22 1.10 1.39 1.24</td>
</tr>
<tr>
<td>Exact $\tau = 0.75$</td>
<td>3944 6763 62274 DNF</td>
<td>1.22 1.10 1.39 1.25</td>
</tr>
<tr>
<td>Exact $\tau = 0.5$</td>
<td>3943 6762 62255 DNF</td>
<td>1.22 1.10 1.39 1.25</td>
</tr>
<tr>
<td>Exact $\tau = 0.25$</td>
<td>3944 6762 62264 DNF</td>
<td>1.22 1.10 1.39 1.25</td>
</tr>
<tr>
<td>Approx $b = 8, r = 16$</td>
<td>99 113 349 2782</td>
<td>1.22 1.34 1.97 8.42</td>
</tr>
<tr>
<td>Approx $b = 16, r = 8$</td>
<td>104 117 362 2906</td>
<td>1.32 1.53 2.53 12.31</td>
</tr>
<tr>
<td>Approx $b = 32, r = 4$</td>
<td>117 128 389 3154</td>
<td>1.49 1.75 3.60 19.29</td>
</tr>
<tr>
<td>Approx $b = 64, r = 2$</td>
<td>154 169 445 3709</td>
<td>1.60 2.12 5.47 27.56</td>
</tr>
<tr>
<td>Approx $b = 2, r = 8$</td>
<td>36 31 102 715</td>
<td>1.07 1.10 1.26 3.01</td>
</tr>
<tr>
<td>Approx $b = 4, r = 4$</td>
<td>38 32 106 753</td>
<td>1.09 1.13 1.39 3.82</td>
</tr>
<tr>
<td>Approx $b = 8, r = 2$</td>
<td>44 37 114 840</td>
<td>1.10 1.17 1.62 4.84</td>
</tr>
</tbody>
</table>

### 4.6.2.2 Total time

The nature of approximation is a trade-off between time and accuracy. The more time you take, the better your accuracy. However, compared to running exact methods, approximation should necessarily require less time to complete. Table 4.2 shows the total time required by the exact and approximate methods. Two different MinHash signature lengths are tested, and within each we vary the number of bands used for indexing.

In every case, regardless of signature size or number of bands, the run time is orders of magnitude lower using our approximation. Using length 128 signatures, approximation yielded average speedups of 33x, 51x, and 161x on Enron, Movielens, and IMDB, respectively. Using
length 16 signatures, approximation yielded average speedups of 134x, 271x, and 775x on Enron, Movielens, and IMDB, respectively. For DBLP, the exact methods were not able to complete in 24 hours, in fact processing less than half of the total stream (see Figure 4.6d). Within any fixed signature length, the time required is proportional to the number of bands used for indexing, as shown in Section 4.5. The number of hashes computed for storing a hyperedge in the index scales one-to-one with the number of bands in the signature.

![Enron Memory Usage](image)

(a) Enron Memory Usage

![IMDB Memory Usage](image)

(b) IMDB Memory Usage

![Movielens Memory Usage](image)

(c) Movielens Memory Usage

![DBLP Memory Usage](image)

(d) DBLP Memory Usage

**Figure 4.7** Memory used by each model over time. The exact models use the same amount of memory, so one time series line is not visible. “Batches” are used for illustrative purposes only, each batch representing the processing of 1,000 hyperedges.

### 4.6.2.3 Memory usage

We showed in Section 4.5 that the memory scales linearly with the size of the stream. This is true for both the approximate model and exact model. Figure 4.7 and Table 4.2 show memory
usage results for both models. Clearly the approximate model uses substantially more memory in our experiments than the exact model. However, in our experiments we stored each key using a Python string object of variable length. Had we used a 4 byte integer as recommended, the memory consumption could be less than that of the exact model, which stores entire hyperedges, depending on the choices of signature length and band sizes.

As expected, different choices of \( b \) and \( r \) results in different memory usage. As \( b \) increases, the number of keys stored in the LSH index commensurately increases. For example, in DBLP the max usage when using \((b, r) = (8, 16)\) is 8.42GB. When using \((64, 2)\) this jumps up to 27.56GB.

### 4.6.3 Identifying Injected Outliers

As no ground-truth data is available for testing the precision of our methods, we must manually create a ground-truth dataset. We follow a method outlined in [10] and inject outliers into four real-world datasets. Artificial outliers are injected by sampling two vertices from a uniform distribution and adding an edge between them. Artificial outliers are injected by first sampling a size \( q \) for the hyperedge, drawn from the empirical distribution of the hyperedge cardinalities in the dataset itself, then sampling \( q \) vertices uniformly at random to be incident. This generation opposes the preferential attachment model, where vertices are connected with probabilities proportional to their degrees and, as such, contradicts the shown power law distribution of the vertex degrees and edge multiplicities [57]. Moreover, the uniform distribution can be shown to be the optimal choice for maximizing the worst case detection rate among all possible anomalous distributions [56, 156, 159]. In total, six types of anomalous datasets were created by varying the percent of outlier hyperedges in the stream, as well as their location (for full details see Section 4.6.1).

We measure the precision of our method by comparing the area under the curve (AUC) of the receiver operating characteristics (ROC) curve for various sketch parameter values. The ROC curve is formed by plotting the true positive rate (TPR) against the false positive rate (FPR) for a range of outlier threshold values. For calculations, we consider the positive class the outlier hyperedges we manually injected, and the rest of the stream the negative class. The TPR is proportional to the number of edges in the positive class that were correctly classified (labeled) as outliers, and the FPR is proportional to the number of edges in the negative class that were falsely classified as positives (i.e. normal edges being classified as outlier edges). The AUC is thus in the range \([0, 1]\), where 1 is a perfect classifier. Table 4.3 shows the results.

We first consider the results for the exact model. One trend that stands out is that the stricter the hyperedge match must be, the worse the performance. Hard matching, \( \tau = 1 \), yields the lowest AUC for every dataset, from 0.514 on IMDB to 0.683 on Enron. Using soft matching with a similarity threshold of 0.25 the AUC jumps up to .672 on IMDB, and 0.974 on
Table 4.3 ROC AUC for each method and dataset. Datasets are labeled E = Enron, M = Movielens, I = IMDB, D = DBLP. Dataset suffix indicates injection type: 1 percent outliers, 5 percent outliers, or Bursty outliers.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Exact Model</th>
<th>Approximate Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>0.683</td>
<td>0.761</td>
</tr>
<tr>
<td>E5</td>
<td>0.683</td>
<td>0.761</td>
</tr>
<tr>
<td>EB</td>
<td>0.683</td>
<td>0.761</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>0.568</td>
<td>0.589</td>
</tr>
<tr>
<td>M5</td>
<td>0.568</td>
<td>0.589</td>
</tr>
<tr>
<td>MB</td>
<td>0.568</td>
<td>0.589</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>0.514</td>
<td>0.517</td>
</tr>
<tr>
<td>I5</td>
<td>0.514</td>
<td>0.517</td>
</tr>
<tr>
<td>IB</td>
<td>0.514</td>
<td>0.517</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>0.640</td>
<td>0.642</td>
</tr>
<tr>
<td>D5</td>
<td>0.641</td>
<td>0.642</td>
</tr>
<tr>
<td>DB</td>
<td>0.640</td>
<td>0.642</td>
</tr>
</tbody>
</table>

Enron. This relaxation producing better results suggests that vertices tend co-occur with their neighbors regularly, but are not limited to interacting with only their existing neighborhood. The significantly lower AUC for IMDB, regardless of method used, is likely a result of the generative process for the stream. Enron captures a professional communication network, which is very stable in its behavior, where IMDB has more one-off interactions and low degree vertices due to the fluidity of the acting roles and degree of difficulty to obtain them. Lastly, the type of injection does not seem to affect the performance of the models, each having almost identical AUC within a given dataset.

Many of the observations about the exact model hold for the approximate model as well. In particular, the AUC for IMDB is also significantly lower, and within a dataset the injection method does not matter. Similar to how lower similarity thresholds yield increased accuracy, using more bands of shorter length increases the accuracy in the approximate model. Intuitively this is because more bands means each hyperedge is hashed to more buckets, effectively lowering the similarity required for two hyperedges to match. For more technical details on how the similarity, see [30, 102]. There are two key results shown in Table 4.3. The first is the relative performance of using length 128 signatures and length 16 signatures. The models produce similar AUC scores for every dataset, but the run time using length 16 signatures is 3-5x faster than using length 128. The second is that both signature lengths performed equally as well as the exact version, but ran in a fraction of the time.
4.7 Conclusion

In this chapter we propose the first approach for performing outlier detection in hyperedge streams. This approach complements the existing body of work, which focuses on static graphs, or restricts relationships (edges) to be pairwise. Our model directly incorporates the temporal dynamics of a generative process, and allows high order relationships to be modeled explicitly. Additionally, we describe an approximation scheme that allows us to vastly speed up the analysis. Experimental results show that both the exact and approximate methods are highly capable of detecting outliers in real-world datasets, and the approximation scheme yields substantial speedups.
CHAPTER 5

FRAUDULENT PATTERN MINING IN THE BITCOIN DIRECTED HYPERGRAPH

5.1 Introduction

In the preceding chapters we have looked at increasingly complex edge streams, focusing on identifying outlier objects in a stream. This chapter shifts focus from individual edges to considering patterns of edges. Specifically, we analyze, theoretically and empirically, 2-patterns in directed hypergraphs (dirhypergraphs). Given the higher order relationships that directed hyperedges, which we will call hyperarcs, are capable of modeling, it is perhaps obvious that the ways multiple hyperarcs can interact is much more combinatorially interesting than in graphs with binary edges. We will theoretically explore the possible 2-patterns in directed hypergraphs, and draw comparisons with graphs when appropriate. From this foundation, we propose a specific 2-pattern the potential manifestation of money laundering or transfer in the Bitcoin cryptocurrency network. A case study of the Bitcoin network is performed, with the goal of understanding how real-world fraudulent behavior may be detectable using graph theoretical methods. Our key insight is to focus on exchanges in the network, which are the entrance and exit from the Bitcoin world.
The major contributions of this chapter can be summarized as follows:

1. **New Model for Bitcoin.** We propose and formalize a dirhypergraph model for the Bitcoin network (Section 5.3). This is distinguished from previous analyses [145, 150] which use graph models with strictly binary edges, whether at the address or tx level.

2. **Dirhypergraph Motifs.** Theoretical and empirical consideration of motifs, specifically 2-patterns, in dirhypergraphs (Sections 5.3, 5.4, and 5.5).

3. **Extensive Exploration and Experiments.** We identify several patterns in the behavior of exchange-owned addresses that differ from non-exchange addresses (Section 5.4). From this, we explore the possibility of applying machine learning techniques to classify latent attributes of addresses (Section 5.5).

### 5.2 Background

Bitcoin is a decentralized virtual currency proposed by Satoshi Nakamoto in 2008 [123]. Rather than relying on a centralized authority (e.g. a bank) to store balances and execute transactions, bitcoin uses a distributed cryptographic ledger to store a complete transaction history and a peer-to-peer protocol to execute transactions. Bitcoin transactions occur between pseudonymous identities and require no knowledge of the identities of other involved parties. An identity, which we will also refer to as an address, is simply the public key of a personally generated asymmetric key pair. As such, an individual may create as many addresses as desired, none of which must be linked to a person’s identity.

Bitcoin’s decentralization makes it difficult to regulate and investigate by law enforcement. This represents a vulnerability in government anti-money laundering (AML) efforts [59]. Conventional AML efforts focus on the Know-Your-Customer (KYC) process, in which banks and other financial services must verify the identity of their customers, monitor transactions, and report suspicious behavior to government entities. As such, government AML and KYC efforts utilize perfect knowledge of identity but incomplete knowledge of financial transactions, which remains in the control of the banks [120, 121]. In contrast, law enforcement generally has no knowledge about bitcoin user identities to use in detecting anomalous behavior, but access to the blockchain grants complete knowledge of transactions. This motivates the desire to detect money laundering through techniques that do not rely on identity information, such as transaction or user patterns. In particular, patterns centered around exchanges are important, as they provide arguably the most important link between Bitcoin and fiat currency networks. Moreover, exchanges are navigating evolving legal precedent to be AML compliant [15, 46]. In 2015, FinCEN fined Ripple Labs in the first act of civil enforcement against a Virtual Currency Exchange for failing to implement a proper AML program [176].
5.3 Bitcoin Transaction Motifs in a Directed Hypergraph

Bitcoin transactions have a natural graphical structure, one form of which is shown on the left of Figure 5.1. Vertices are transactions $E_0, \ldots, E_3$, while arcs model inputs and outputs labeled by the Bitcoin address $a_1, \ldots, a_6$, weighted by the quantity. We note some common activities such as change making and aggregation. Coinbase transactions are indicated by the vertex SRC in blue, while unspent transaction outputs (UTXOs) are combined into a single sink vertex.

Figure 5.1 (Left) Bitcoin transactions as a labeled multigraph. (Right) Bitcoin transactions as a bipartite multigraph.

But in analytical tasks, such as detecting money laundering, it is perhaps more important to focus on addresses (arc labels $a_i$) than the transactions $E_j$. While discouraged in Bitcoin, reuse of addresses is allowed and somewhat common. To treat addresses as “first class objects” we create new vertices for each unique address, producing the bipartite graph structure on the right of Figure 5.1. Square vertices are transactions $E_j$, while the circles are addresses $a_i$ (addresses currently with an UTXO are red). Input arcs from addresses to transactions are now distinguishable from output arcs from transactions to addresses, and address reuse can create looping structures, such as shown in change-making back into $a_3$ as an output of $E_2$.

We consolidate by combining quantities on multi-arcs, producing the directed hypergraph [18, 62] in Figure 5.2. Dirhypergraphs are characterized as directed bipartite graphs with two kinds of vertices, with connections only between vertices of different types, but possibly multiple inputs from and outputs to each. For us, transaction (square) vertices act as directed hyperarcs. Where an arc in a graph connects a single tail vertex to a single head vertex, hyperarcs connect multiple input (tail) addresses to multiple output (head) addresses (round vertices).

Hypergraphs and directed hypergraphs are well known in math and computer science, and can provide significant advantages over regular graphical structures when data are complex. Here the data complexity lies in the allowance of multiple inputs and multiple outputs for each transaction. Identifying subgraphs indicating potential laundering suggests the potential
Figure 5.2 Bitcoin transactions as a directed hypergraph.

Figure 5.3 Graph motifs: (Far Left) The single undirected graph 2-motif (above) with its three directed motifs $\alpha$, $\beta$, and $\gamma$ (below), including the two isomorphic $\alpha, \alpha'$ patterns. (Right) The three undirected 3-motifs and their directed versions.
significance of hypergraph motifs. In network analysis, motifs are small subgraphs which are represented with statistical significance [117]. This work appears to be the first to consider dirhypergraph motifs, which we generalize directly from graph motifs. Figure 5.3 shows all the undirected and directed motifs for two and three edges/ars. A $k$-motif is one possible way that $k$ connected (intersecting) edges can be structured, with a range of possible numbers of vertices sitting as tails and heads of the $k$ edges.

The simplest dirhypergraph Bitcoin pattern is the 2-motif,$^1$ illustrated on the left side of Figure 5.4 as a dirhypergraph pattern. As in Figure 5.2, the two transactions $E_1, E_2$ are square vertices, and addresses are circles (circle color will be addressed below). Note that any particular address can sit on either the tails (inputs) or heads (outputs) of any transaction, and indeed, more than one transaction.$^2$

![Figure 5.4](image)

**Figure 5.4** A 2-motif in a dirhypergraph: Two transactions $E_1, E_2$ with sets of tail and head vertices $T_1, H_1, T_2, H_2$ respectively. (Left) Generic. (Center) A linear STB. (Right) A circular STB.

Formalizing 2-motifs, assume a non-empty finite set of vertices $A = \{a_i\}$ and two hyperarcs $E_j = \langle T_j, H_j \rangle, j = 1, 2$, with tails and heads $T_j, H_j \subseteq A$ (nonempty). The set $M = \{E_1, E_2\}$ is a 2-motif if there is at least one pair of tails and heads in each hyperarc which intersect, that is, if $\bigcup_{X,Y \in \{H,T\}} X_1 \cap Y_2 \neq \emptyset$. The potential intersections are detailed in Table 5.1. For example, an address sits on a $\gamma$ pattern if it is in both $T_1$ (the blue vertices in Figure 5.4) and $T_2$ (the green), that is, in the tails of (inputs to) both transactions (e.g. the lowest left address in Figure 5.4). The analogous graph pattern $\gamma$ is shown in Figure 5.3, with the two edges pointed outward. Note that in comparison with Figure 5.3, here we also allow self-loops identified in the $L_1, L_2$ patterns.

The left of Figure 5.5 is an abstraction of a 2-motif, where each circular vertex represents one of the patterns in Table 5.1, standing in for the entire set of addresses playing that role.$^3$

---

$^1$Terminologically, we can call these hypermotifs or hypergraph motifs, but for simplicity here we will just call them motifs.

$^2$This is significant as most other work with dirhypergraphs, indeed every work we looked at, requires the head and tail vertex sets be disjoint. This has a large impact on the range of motifs possible.

$^3$We include addresses in exactly one intersection, ignoring addresses in only a tail or head of one of the
Table 5.1 Participation of addresses in a 2-motif.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$H_1 \cap T_2 \neq \emptyset$</td>
<td>Forward 2-chain</td>
</tr>
<tr>
<td>$\alpha'$</td>
<td>$T_1 \cap H_2 \neq \emptyset$</td>
<td>Reverse 2-chain</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$H_1 \cap H_2 \neq \emptyset$</td>
<td>Inward 2-star</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>$T_1 \cap T_2 \neq \emptyset$</td>
<td>Outward 2-star</td>
</tr>
<tr>
<td>$L_1$</td>
<td>$T_1 \cap H_1 \neq \emptyset$</td>
<td>Self-loop on $E_1$</td>
</tr>
<tr>
<td>$L_2$</td>
<td>$T_2 \cap H_2 \neq \emptyset$</td>
<td>Self-loop on $E_2$</td>
</tr>
</tbody>
</table>

The right is the same abstraction, but now with the counts of the number of addresses in each role for transactions between Jan 12 2015 and April 21 2015.\(^4\)

![Figure 5.5](image)

**Figure 5.5** (Left) Generic 2-motif. (Right) Instantiated with counts for days 2200-2299 of our data. For details about our dataset, see Section 5.4.

Beyond just identifying dirhypergraph motifs, we are interested in motifs which may or may not involve certain addresses, in our case, exchanges, and their distribution within certain kinds of patterns. In Figure 5.4, exchanges are shown as black addresses. What we call a **short thick band (STB)** is a pattern where a quantity of Bitcoin is purchased from fiat currency, held for a while as Bitcoin, and then converted back to fiat currency. When an STB moves Bitcoin from one exchange address to a different one, we can call it **linear**; and when it returns it to the same exchange address, **circular**. More specifically, STBs are 2-motifs where:

- Two transactions intersect in an $\alpha$ or $\alpha'$ chain only;
- An exchange is included in both an input of the first and an output of the last transaction; but

\(^4\)Note the similarities of the counts for $\alpha$ and $\alpha'$, on the one hand, and $L_1$ and $L_2$, on the other, due to isomorphism with respect to the ordering of $E_1$ and $E_2$. 

transactions, and also addresses in *more than* two intersections.
• No exchange is an intermediate address in the transaction.

STBs could exist for many reasons, including financial speculation, simple user convenience, repeated purchases, remuneration, remittance, or fund management. Our interest is considering the hypothesis that STBs could be used as a potential laundering pattern. Moreover, we recognize that as a laundering pattern, it would not be very extensive. In this work we are beginning with the simplest possible such pattern.

To formalize STBs, call a motif “pure” if only one pattern from Table 5.1 is present (this is not the case in the left side of Figure 5.4), and otherwise “mixed”. Then denote $e(a \in A)$ to mean that $a$ is an exchange, and $e(X \subseteq A)$ to mean that $X$ has an exchange: $\exists a \in X, e(a)$. We then can define an STB as follows.

**Definition 5.1 (STB).** A 2-hypermotif $M = \{E_1, E_2\}$ is a linear STB if one and only one of the following holds:

1. It is a pure $\alpha$ 2-motif with $e(T_1) \land e(H_2) \land \neg e(H_1 \cap T_2)$; or
2. It is a pure $\alpha'$ 2-motif with $e(H_1) \land e(T_2) \land \neg e(T_1 \cap H_2)$.

$M$ is a circular STB if one and only one of the following holds:

1. It is a mixed linear $\alpha$ and $\alpha'$ 2-motif with $e(T_1 \cap H_2) \land \neg e(H_1 \cap T_2)$; or
2. It is a mixed linear $\alpha'$ and $\alpha$ 2-motif with $e(H_1 \cap T_2) \land \neg e(T_1 \cap H_2)$.

Note that no STB can have a self-loop. But because of the $\alpha, \alpha'$ isomorphism noted above, it is sufficient to assume that a linear STB is a pure $\alpha$ pattern, and a circular STB is a mixed $\alpha, \alpha'$ pattern, with

$$e(T_1) \land e(H_2) \land \neg e(H_1 \cap T_2), \quad e(T_1 \cap H_2) \land \neg e(H_1 \cap T_2)$$

respectively. Figure 5.4 shows a linear (center) and circular (right) STB.

### 5.4 Descriptive Statistics

Given the nature of exchanges, and their primary function of converting between bitcoin and other currencies (including fiat and other alt coins), we question whether exchange addresses exhibit a different type of behavior from address owned by regular users of the network.

We downloaded the Bitcoin blockchain data using the Bitcoin Core Client,\footnote{https://bitcoin.org/en/download} and built a custom parser to convert the raw data into the dirhypergraph structure described in Section 5.3.
We used data from the first transaction in the network up to April 22 2015, encompassing 72.7M unique addresses (vertices), involved in at least one of 66.3M transactions (hyperarcs) in our dirhypergraph. Addresses known to be exchanges were drawn from the WalletExplorer listing, call these “labeled”. Some exchanges are associated with several wallets (“current”, “output”, “old”). The full list we use is shown in Appendix B. While labeled addresses are presumed to be actual exchanges, the number of exchange addresses which are not listed as such is hard to judge for many reasons. Indeed, the WalletExplorer listings began on April 23 2011, while we know that exchanges have been around since 2010. Additionally, Mt. Gox, a substantial contributor over that time, was not included. There are still 2.44M labeled addresses (3.36 % of the total addresses), and 6.76M transactions involving an exchange (10.2 % of total transactions). Daily activity is summarized in Figure 5.6, with addresses involved in several transactions in a single day counted once.

Figure 5.6 Daily activity in each dirhypergraph.

Since our dirhypergraph presents as a bipartite graph of addresses and transactions, the in-degree of an address is actually the number of transactions on which an address serves as an output, and vice versa for out-degree. While the in- and out-degree distributions for both address types follow a power law, with the distributions having no significant difference using a 2-sample KS test, Table 5.2 shows several interesting aspects. Where 4.7% of unlabeled addresses are sinks—simply accumulating bitcoin and never redistributing it, resulting in an out-degree of zero—this drops to < 0.1% for labeled addresses. Moreover, labeled addresses are more likely to have equal and positive in- and out-degrees. This behavior for labeled addresses is consistent with the use and function of exchanges, and for unlabeled addresses it is consistent with previous results [150], although to a much lesser extent. We attribute the decline in the proportion of sink addresses to the general growth of Bitcoin, but more importantly the sustained trading phase [95] it has been in, dwarfing activity in the initial phase.

Table 5.2 Comparing labeled and unlabeled address’ degrees and weights.

<table>
<thead>
<tr>
<th>Query</th>
<th>Labeled</th>
<th>Unlabeled</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-degree &gt; 0, out-degree = 0</td>
<td>2,123</td>
<td>3,297,725</td>
</tr>
<tr>
<td>In-degree &gt; 0, out-degree &gt; 0</td>
<td>2,435,472</td>
<td>66,914,170</td>
</tr>
<tr>
<td>In-degree = out-degree &gt; 0</td>
<td>2,356,530</td>
<td>64,305,459</td>
</tr>
<tr>
<td>In-weight &gt; 0, out-weight = 0</td>
<td>2,123</td>
<td>3,297,195</td>
</tr>
<tr>
<td>In-weight &gt; 0, out-weight &gt; 0</td>
<td>2,435,472</td>
<td>66,914,166</td>
</tr>
<tr>
<td>In-weight = out-weight &gt; 0</td>
<td>2,421,944</td>
<td>65,658,855</td>
</tr>
</tbody>
</table>

Figure 5.7 shows the cumulative distribution of weights in bitcoin (BTC), total on the left and average on the right. Unlabeled addresses have a much better separation between the in- and out-weight, suggesting that non-exchange addresses tend to keep a positive balance of bitcoin while exchanges keep zero, or near-zero, balances. The majority of exchange addresses have both an average and total transaction weight between 0.01 and 0.1 BTC, shown by the large jump in the figure. Roughly 50% of labeled addresses sit in this bucket, compared to about 30% of unlabeled addresses.

We next examine 2-motifs, STBs, and how exchanges are involved in them. If there are \( m \) transactions in a day, then there are at most \( \binom{m}{2} \) possible 2-motifs. Of our 40.0B 2-motifs, 10.3B are pure linear \( \alpha \) or \( \alpha' \) patterns, just 42.4M of which involve exchanges. 741K of those are STBs, including 727K linear and 13.4K circular. The volume of 2-motifs precludes the opportunity to do a comparison between STBs and non-STB 2-motifs, so instead we focus on just STBs.

The number and proportion of addresses that are labeled as exchanges on the inputs and
Figure 5.7 Cumulative percent of addresses with total (left) or average (right) input and output weights.

Figure 5.8 Counts (left) and proportions (right) of labeled addresses on the inputs and outputs of STBs.

The number of labeled input addresses ranges from 1 to 635, while output addresses range from 1 to 1937. However, when the two clear outlier STBs are removed, the max number shrinks to 376.

According to the well known heuristic in Bitcoin to group all addresses that are inputs into the same transaction as being owned by the same entity [113, 126, 150], the expected proportion of labeled inputs should be 1, because if a single address is labeled, then all others are labeled as a consequence. Orders of magnitude more STBs do in fact have all inputs labeled, but many

---

7Recall that this is at the address level, and each exchange has a set of addresses they own. The frequency of each exchange (e.g., BTC-e.com) in STBs is shown in Figure B.1 in the Appendix, and is highly correlated with the number of addresses each exchange has, see Figure B.2.
do not. This could be because the WalletExplorer data is incomplete, because it uses a different method for aggregating exchange wallets, or because they should in fact not be labeled. Yet, in every STB with multiple labeled input addresses, every address is owned by the same exchange—that is, every input is from a single exchange label. Interestingly, this is not the case for outputs of STBs. Of the 739.8K STBs, 63.4K (8.57%) have multiple exchange labels in the outputs (“multi-out STBs”), 48.1K (75.9%) of which have exactly two labels. Multiple exchanges on outputs of a single transaction could be due to mining pool payouts, but on outputs of STBs it is more likely indicative of inter-exchange activity.

5.5 Classifying and Labeling Exchange Addresses

While exchange addresses comprise a small percent of the Bitcoin network, they are of growing importance outside of the Bitcoin world, as they provide potentially the only avenue for connecting real-world people with pseudonymous Bitcoin addresses. Being able to identify exchanges in the network is then a critical task, as it enables one to connect transactions or addresses of interest to the point at which they enter or exit the Bitcoin network.

We can leverage the different characteristics of exchange and non-exchange addresses to construct a machine learning model to classify an address as an exchange or not. In these experiments we use data from September 29 2011, roughly 100 days after exchanges first appear in our data, until April 22 2015. For every address we extract a set of features that numerically characterizes it, e.g. out-degree, total in-weight (full feature set described in Appendix B). Addresses are then assigned a class label corresponding to whether or not they are labeled as an exchange. The goal of the model is to learn a set of features and weights that can accurately discriminate between the two classes. Given the immense class imbalance (far fewer exchange addresses), we randomly sample an equal number of labeled and unlabeled addresses for training and testing the model. To account for the random sampling, 10 independent trials are run, and average results are reported.

Five different classifiers’ results are summarized in Table 5.3. AdaBoost and random forests perform the best, and are far superior to the remaining three, both yielding an F1 score of over 0.99. In the case of random forests, on average only 2,587 out of 972,866 test addresses were incorrectly predicted, falsely classifying 1,190 non-exchanges as exchanges, and 1,397 exchanges as non-exchanges. Moreover, the incredibly low variance of these models indicates they are much more robust than the others, performing well across all random samples.

Equally important, or perhaps even more important, than achieving such a high accuracy is understanding what it is about exchange addresses that facilitates the result. One way to quantify this is looking at the “feature importance” values that are calculated by the classifier.

---

8 All experiments were run using Python 2.7 and the scikit-learn and numpy packages.
Table 5.3 Results for exchange address classification. All results shown are mean $\pm$ std over the 10 runs.

<table>
<thead>
<tr>
<th>Model</th>
<th>F1</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest</td>
<td>0.9973 $\pm$ (0.0001)</td>
<td>0.9976 $\pm$ (0.0001)</td>
<td>0.9971 $\pm$ (0.0001)</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>0.9944 $\pm$ (0.0001)</td>
<td>0.9974 $\pm$ (0.0001)</td>
<td>0.9915 $\pm$ (0.0003)</td>
</tr>
<tr>
<td>Linear SVM</td>
<td>0.8291 $\pm$ (0.0833)</td>
<td>0.8396 $\pm$ (0.1514)</td>
<td>0.8573 $\pm$ (0.1209)</td>
</tr>
<tr>
<td>Perceptron</td>
<td>0.2075 $\pm$ (0.3029)</td>
<td>0.3034 $\pm$ (0.4557)</td>
<td>0.2210 $\pm$ (0.2053)</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>0.0014 $\pm$ (0.0001)</td>
<td>0.0007 $\pm$ (0.0001)</td>
<td>0.2755 $\pm$ (0.0304)</td>
</tr>
</tbody>
</table>

The top 5 features and their importance in the random forest model are: (1) # sibling exchanges (0.613); (2) # successor exchanges (0.184); (3) # predecessor exchanges (0.072); (4) # siblings total (0.044); (5) total out-weight (0.015). It is not surprising that, by far, the most important feature is the number of exchange siblings. Figure 5.9a shows the substantial difference in the distributions for exchange siblings. Again, according to the common address grouping heuristic, if you are siblings with numerous exchange addresses then it is likely you are also an exchange address. Moreover, it is likely that you are an exchange address owned by the same exchange that your siblings are (c.f. Section 5.4). From a network science perspective, homophily [112] tells us that vertices of one type tend to interact with vertices of the same type, i.e. exchanges link with exchanges.

Figure 5.9 Distribution of how many exchange addresses siblings (left) or total siblings (right) each address has. These distributions were drawn from a random sample of 100K exchanges and 100K non-exchanges.

---

For an address $a$, siblings are addresses that have been a co-input or co-output, successors are addresses that have been an output when $a$ was an input, and predecessors are addresses that were an input when $a$ was an output.
Incorporating features related to the exchange labels clearly produces high quality results. However, it may restrict the capability of our model, failing to generalize well to new data which is not labeled, or handling incomplete labeling as we have in our dataset. To test this a second set of experiments is performed, identical to those described above except all features related to exchange labels are removed. Table 5.4 summarizes the new results.

<table>
<thead>
<tr>
<th>Model</th>
<th>F1</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest</td>
<td>0.8200 +- (0.0004)</td>
<td>0.8218 +- (0.0006)</td>
<td>0.8183 +- (0.0004)</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>0.7941 +- (0.0012)</td>
<td>0.8264 +- (0.0033)</td>
<td>0.7643 +- (0.0018)</td>
</tr>
<tr>
<td>Linear SVM</td>
<td>0.3052 +- (0.2619)</td>
<td>0.3488 +- (0.3775)</td>
<td>0.5179 +- (0.1710)</td>
</tr>
<tr>
<td>Perceptron</td>
<td>0.1349 +- (0.2683)</td>
<td>0.1998 +- (0.3989)</td>
<td>0.1849 +- (0.1886)</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>0.0014 +- (0.0001)</td>
<td>0.0007 +- (0.0001)</td>
<td>0.3056 +- (0.0266)</td>
</tr>
</tbody>
</table>

Removal of the exchange features has an obvious negative impact on the accuracy of the classifiers. Random forest F1 score drops to 0.82 (a decrease of about .18), with the average number of incorrectly classified addresses increasing from 2587 (0.266%) to 175956 (18.086%). Similar to the runs that included exchange features, the average variance for the random forest classifier was very low. With the removal of exchange related features, structural features rose in importance. The new top 5 features and weights are: (1) # siblings (0.255); (2) total out-weight (0.232); (3) total in-weight (0.217); (4) # successors (0.092); (5) # predecessors (0.082). The former fourth and fifth ranked features are now the top two, and in conjunction with the total in-weight represent the majority of the discriminatory power. The top three now have a very equal share of importance, indicating that the model relies on information from each of them instead of a single dominating feature. Figure 5.9 shows the distribution of the number of siblings for both exchange and non-exchange addresses. The distributions for less than 100 siblings are easily separable, but become much more intertwined when considering addresses with 100 or more siblings.

As we note in Section 5.4, the list of exchange addresses we have is incomplete. However, it is impossible to know exactly how incomplete the list is – whether we have 10% of the exchange addresses or 90%. A natural next question, then, is to try to classify all of the unlabeled addresses using our models constructed in the previous experiments.

All unlabeled addresses not used in training the classifiers were run through both random forest models and predicted as an exchange address or not (Table 5.5). The two classifiers
yielded drastically different results. Using exchange label features, a mere 0.28% of the unlabeled addresses were labeled as exchanges. Conversely, 18.17% of the addresses were labeled as exchanges using the purely structural features. If instead of omitting the training addresses from the results we include them, the percent predicted raise to 3.98% and 21.87%. As 3.36% of the addresses are labeled from our ground truth data, this result is expected.

It is likely that 18% is a much better estimate for the true exchange address percent than 0.28%. The absence of Mt. Gox (among others) from our label data, and its historical dominance in Bitcoin, indicates that we are missing a large number of exchange addresses. Moreover, the extremely high accuracy combined with the extremely low prediction of unlabeled addresses of the first model suggests that the first classifier overfit the training data, exploiting the label features and becoming too reliant. Structural features, which we have perfect knowledge of for all addresses, are much more reliable and generalizable.

Table 5.5 Percent of addresses classified as exchanges.

<table>
<thead>
<tr>
<th></th>
<th>With Label Features</th>
<th>Without Label Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>All addresses</td>
<td>3.98%</td>
<td>21.87%</td>
</tr>
<tr>
<td>Unlabeled addresses</td>
<td>0.28%</td>
<td>18.17%</td>
</tr>
<tr>
<td>Middle addresses 1-out STBs</td>
<td>1.34%</td>
<td>48.35%</td>
</tr>
<tr>
<td>Middle addresses multi-out STBs</td>
<td>0.68%</td>
<td>52.09%</td>
</tr>
</tbody>
</table>

Our initial proposition of STBs as a laundering pattern stems from a user activity view of the network: a user receives bitcoin from an exchange, then converts it back into fiat currency, with the hope of obfuscating any money trail. From this perspective, addresses in the middle of an STB – which by definition cannot be labeled as an exchange – should be less likely to be predicted as an exchange than a randomly chosen unlabeled address. But (see Table 5.5) addresses in the middle of an STB are 2-3x more likely to be classified as an exchange than an a random unlabeled address. This directly contradicts our hypothesis, and instead is highly suggestive of lots of inter-exchange activity taking place. Self-churn [113] i.e. change-making is likely why an exchange address would be in the middle of what would otherwise be an STB. For example, an exchange $E_1$ sends bitcoin to one of its customers, making change for itself with the excess bitcoin in the transaction, and then another exchange $E_2$ buys bitcoin from $E_1$, creating a 2-motif with exchanges on the input, middle, and output.
5.6 Conclusions and Future Work

In this chapter we make a first attempt at statistical and machine learning approaches that may be of interest in identifying laundering patterns, latent attribute classification, and discriminatory analysis. Directed hypergraphs are a sound model for transactions, and exchanges exhibit several patterns that are distinct from general address behaviors, as also shown in previous work. Our machine learning models are capable of labeling addresses as being owned by exchanges or not with very high accuracy, even when restricted to purely structural features. STBs are proposed as a potential laundering pattern, and shown to have a high degree of filtering when compared to the number of general 2-motifs in the network. Finally, we showed that middle vertices in STBs are much more likely to be classified as an exchange, indicating that there is a large amount of inter-exchange activity taking place.

Obvious areas of improvement include a much better label set, including Mt. Gox and generally being of higher fidelity. Similarly, an obvious area of expansion is to move beyond 2-motifs to 3-motifs, and consider triangular and other patterns involving three transactions. We have began the mathematical exploration of the 3-motif in directed hypergraphs, and it is somewhat complicated combinatorially, but manageable. While we use learning to label an address as an exchange or not, the general tasks of latent attribute learning and discriminatory feature analysis impose no such constraint. A variety of customary labels may be of interest [113] – “mining pool”, “wallet”, “exchange”, “vendor”, “gambling” – in addition to your own personal labels – “suspicious”, “country X”. It is also not necessary to constrain the analysis to a single label at a time, but instead use multi-class classification models to predict from a set of labels. Moreover, instead of using a supervised learning approach where we assume our label sets are complete, we could explore methods such as PU Learning [104, 107], which account for imperfect data.

In addition to expanding the possible labels, the structure of interest could be expanded as well. Instead of looking at single addresses, transactions, or chains of transactions that form a higher level pattern, could be considered. For example, Möser et al. [120] show that some mixing services leave a distinct transactional pattern as a result of their mixing algorithms. Models for identifying similar patterns could be constructed using the hand curated transactions found in [120].
Graphs have become a ubiquitous way to model complex systems and relationships. Mining interesting structures in the graph model leads to insight about the underlying system. Developing efficient methods for mining large dynamic graphs has become extremely important, as the scale of data continues to grow. The majority of work to date has been focused graph streams and exact computation, which is quickly becoming infeasible. In this dissertation we focused on developing methods for efficiently mining outliers in various dynamic graph models, showing that approximation and clever use of data structures can greatly reduce the computational requirements.

Our fundamental proposition, based on opportunities identified by the survey done in Chapter 2, was that by transitioning to an edge-centric analysis, e.g. edge streams, and leveraging approximation we would be able to greatly enhance the feasibility of real-time graph anomaly detection. We approached this problem from several different angles. In Chapter 3 we propose the first algorithm for mining outliers in an edge stream data model, and we do so efficiently by storing the salient information in CountMin sketches. Chapter 4 provides a generalization of Chapter 3, by looking at hyperedge streams. This transition from edges to hyperedges required us to employ a different approximation approach, instead using MinHash and LSH. Finally, Chapter 5 is a case study in mining potential fraud patterns in the dynamic bitcoin directed hypergraph.
6.1 Future Work

Mining anomalies in graph models, in particular in dynamic graph models, is a rapidly expanding field of research. We conclude this dissertation by describing several interesting potential directions for future work.

First, additional attribute information could be incorporated into the graph model. For example, Bello et al. [24] use author research area as an attribute in the DBLP graph when mining communities. All of the graphs we used were “plain”, meaning the vertices and edges had no attributes associated with them. Many real-world systems provide a wealth of additional information that could be used to better identify structural anomalies, and even define new types of anomalies. Already several research groups [49, 103, 127] have started to consider this problem in a static setting, for example identifying contextual anomalies [103]. A related concept is that of heterogeneous graphs, which have classes of vertices and edge types. For example, the DBLP graph could be modeled as a heterogeneous graph by having vertex types \{author, paper, conference\}, and corresponding edge types such as “published in”. One initial work in this direction is by Chen and Neill [39] on using non-parametric scan statistics for event detection.

Second, distributed and parallel algorithms for anomaly detection are going to be essential as the amount of data available continues to grow. For graph mining, frameworks such as Pegasus [50], Pregel [110], and Giraph [19] provide an easy to use API for performing many basic graph queries, such as PageRank and shortest paths. One of the biggest challenges faced by these frameworks is how to partition the network, affecting the information that each compute node has to work with. One potential approach to alleviate this is to borrow from Chapter 3 and use sketches. Each compute node, in real time, can process the edges of the graph that they receive, and at fixed intervals sync up their sketches with the rest of the compute network. CM sketches are one example of a data structure capable of this.

Third, considering how to reset, decay, or otherwise age the model. As we continue to gather new information, graph edges or otherwise, the old information can become stale. One very important question to ask, is how to identify stale data and remove it, or how to globally account data becoming stale. A straightforward approach is to simply decay all values over time. For example, in Chapter 3, every $T$ timesteps we could halve the values in the CM sketches, similar to the approaches taken in [105, 183]. Another potential avenue is to maintain an ensemble of models, where every $T$ timesteps a new model is created. Newer members would contain only recent information, and hence base decisions off that, while older models have a more complete history.

Finally, we could adapt our methods for other graph mining tasks, such as centrality measures and community detection. For example, in Chapters 3 and 4 we store a great deal of the structural information about the graph. As new edges are processed, the local centrality estimations could
be updated for each affected vertex.


99


ANOMALY DETECTION SURVEY

SUMMARY TABLES

**Table A.1** Algorithm Complexity Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>Number of vertices in the graph</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of edges in the graph</td>
</tr>
<tr>
<td>$t$</td>
<td>Number of time steps</td>
</tr>
<tr>
<td>$k$</td>
<td>Number of clusters</td>
</tr>
<tr>
<td>$\mathcal{X}$</td>
<td>Tensor representing the graph</td>
</tr>
<tr>
<td>$\text{nnz}(\mathcal{X})$</td>
<td>Number of nonzero elements in $\mathcal{X}$</td>
</tr>
<tr>
<td>$</td>
<td>\mathcal{X}</td>
</tr>
<tr>
<td>$r$</td>
<td>Size of the largest mode of the core tensor approximation (e.g., in the Tucker decomposition [175])</td>
</tr>
<tr>
<td>$N_{\mathcal{X}}$</td>
<td>Number of modes in $\mathcal{X}$</td>
</tr>
<tr>
<td>$c$</td>
<td>Number of columns sampled for low rank approximation</td>
</tr>
<tr>
<td>$\text{Exp.}$</td>
<td>Exponential time complexity</td>
</tr>
</tbody>
</table>
Table A.2 Summary of Methods. Checkmarks indicate type of anomaly found by each method.
*The complexity stated is, in some cases, a coarse approximation provided for comparative purposes, and is for the entire graph series.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Vertex</th>
<th>Edge</th>
<th>Subgraph</th>
<th>Event</th>
<th>Year</th>
<th>Complexity*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Community</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duan [53]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>2009</td>
<td>$O(n^3t)$</td>
</tr>
<tr>
<td>Aggarwal [3]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>2012</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Chen [41]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>2012</td>
<td>Exp.</td>
</tr>
<tr>
<td>Gupta [76]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>2012</td>
<td>$O(nk^2t)$</td>
</tr>
<tr>
<td>Gupta [75]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>2012</td>
<td>$O(n^k2^t)$</td>
</tr>
<tr>
<td>Chen [42]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>2013</td>
<td>Exp.</td>
</tr>
<tr>
<td>Ji [88]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>2013</td>
<td>$O(n^3t)$</td>
</tr>
<tr>
<td>Rossi [152]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2013</td>
<td>$O(mt)$</td>
</tr>
<tr>
<td>Araujo [17]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>2014</td>
<td>$O(nnz(\mathcal{X}))$</td>
</tr>
<tr>
<td><strong>Compression</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chakrabarti [33]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>2004</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Sun [163]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>2007</td>
<td>$O(tm)$</td>
</tr>
<tr>
<td><strong>Decomposition</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ide [86]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2004</td>
<td>$O(n^2t)$</td>
</tr>
<tr>
<td>Lakhina [99]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2004</td>
<td>$O(tm^2)$</td>
</tr>
<tr>
<td>Sun [166]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2006</td>
<td>$O(nr^3t)$</td>
</tr>
<tr>
<td>Sun [162]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2006</td>
<td>$O(rt</td>
</tr>
<tr>
<td>Sun [164]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2007</td>
<td>$O(nc + c^3)$</td>
</tr>
<tr>
<td>Kolda [93]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2008</td>
<td>$O(</td>
</tr>
<tr>
<td>Tong [173]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2008</td>
<td>$O(mc + c^3)$</td>
</tr>
<tr>
<td>Akoglu [7]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2010</td>
<td>$O(n^2t)$</td>
</tr>
<tr>
<td>Ishibashi [87]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>2010</td>
<td>$O(n^2t)$</td>
</tr>
<tr>
<td>Jiang [89]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2011</td>
<td>$O(nt^2 + n^2)$</td>
</tr>
<tr>
<td>Koutra [96]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2012</td>
<td>$O(nnz(\mathcal{X}))$</td>
</tr>
<tr>
<td>Miller [114]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>2012</td>
<td>$O(mt)$</td>
</tr>
<tr>
<td>Papalexakis [131]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2012</td>
<td>$O(nnz(\mathcal{X}))$</td>
</tr>
<tr>
<td>Miller [115]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>2013</td>
<td>$O(mt)$</td>
</tr>
<tr>
<td>Yu [183]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>2013</td>
<td>$O(n^3)$</td>
</tr>
</tbody>
</table>

108
Table A.3 Summary of Methods. Checkmarks indicate type of anomaly found by each method.
*The complexity stated is, in some cases, a coarse approximation provided for comparative purposes, and is for the entire graph series.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Vertex</th>
<th>Edge</th>
<th>Subgraph</th>
<th>Event</th>
<th>Year</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Distance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pincombe [134]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2005</td>
<td>$O(m^2t)$</td>
</tr>
<tr>
<td>Gaston [65]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2006</td>
<td>$O(n^3t)$</td>
</tr>
<tr>
<td>Li [105]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2009</td>
<td>$O(m^2t)$</td>
</tr>
<tr>
<td>Abello [1]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>2010</td>
<td>$O(mt)$</td>
</tr>
<tr>
<td>Papadimitriou [129]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2010</td>
<td>$O(mt)$</td>
</tr>
<tr>
<td>Arackaparambil [16]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2011</td>
<td>$O(mt)$</td>
</tr>
<tr>
<td>Le [101]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2011</td>
<td>$O(n^2t)$</td>
</tr>
<tr>
<td>Berlingerio [25]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2012</td>
<td>$O(nt \log n)$</td>
</tr>
<tr>
<td>He [81]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2012</td>
<td>$O(n^3t)$</td>
</tr>
<tr>
<td>Malliaros [111]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2012</td>
<td>$O(n^2t)$</td>
</tr>
<tr>
<td>Koutra [58]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2013</td>
<td>$O(mt)$</td>
</tr>
<tr>
<td>Mongiovì [119]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2013</td>
<td>$O(mt + m \log m)$</td>
</tr>
<tr>
<td>Mongiovì [118]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2013</td>
<td>$O(mt \log m)$</td>
</tr>
<tr>
<td><strong>Probabilistic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priebe [136]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2005</td>
<td>$O(n^3t)$</td>
</tr>
<tr>
<td>Huang [85]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>2006</td>
<td>$O(n^3t)$</td>
</tr>
<tr>
<td>Pandit [128]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2007</td>
<td>$O(mt)$</td>
</tr>
<tr>
<td>Hirose [83]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>2009</td>
<td>$O(n^2t)$</td>
</tr>
<tr>
<td>Thompson [171]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>2009</td>
<td>$O(n^2t)$</td>
</tr>
<tr>
<td>Heard [82]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>2010</td>
<td>$O(n^2t)$</td>
</tr>
<tr>
<td>Djidjiev [52]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>2011</td>
<td>$O((n + m)t)$</td>
</tr>
<tr>
<td>Neil [124]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>2013</td>
<td>$O(mn^{k-1})$</td>
</tr>
</tbody>
</table>
Table A.4 Public Datasets; †Data may not already be in network format; *Synthetic instead of real data; ‡Many more datasets are available from the 2008, 2009, and 2010 challenges; See [161] for one example of creating climate networks from raw data.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Network Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNAP</td>
<td>Multiple</td>
<td>A large collection of networks, both static and dynamic, including domains such as network traffic, social networks, and online reviews. See also Newman’s collection.</td>
</tr>
<tr>
<td>VAST 2008*1</td>
<td>Social</td>
<td>A who-calls-whom dataset, recording 10 fictional days of calls and 400 unique cell phones.</td>
</tr>
<tr>
<td>PeMS</td>
<td>Vehicle traffic</td>
<td>Provides up to 10 years of real historical data on the road networks in California.</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>Web links</td>
<td>Provides an expansive Wikipedia data set that has information such as page links, page views, and page revisions.</td>
</tr>
<tr>
<td>Enron</td>
<td>Communication</td>
<td>A who-emails-whom network between about 150 former Enron employees, mostly upper management.</td>
</tr>
<tr>
<td>IMDB</td>
<td>Social</td>
<td>Over 100 years worth of movie data, including titles, producers, and actors.</td>
</tr>
<tr>
<td>DBLP</td>
<td>Co-author</td>
<td>Computer science co-author network.</td>
</tr>
<tr>
<td>US patents</td>
<td>Citation</td>
<td>A list of almost 3 million U.S. patents granted between January 1963 and December 1999, and all citations made to these patents between 1975 and 1999.</td>
</tr>
<tr>
<td>Climate Reanalysis2</td>
<td>Climate</td>
<td>Various climate variables (air temperature, precipitable water) assimilated from 1948 to the present, across the globe in a fixed grid.</td>
</tr>
<tr>
<td>Reality Commons</td>
<td>Communication and social</td>
<td>Multiple datasets, each with information on who-calls-whom and proximity based connections.</td>
</tr>
<tr>
<td>HEP-Th</td>
<td>Citation</td>
<td>Consolidated data from the KDD 2003 datasets for theoretical high energy physics papers.</td>
</tr>
<tr>
<td>Can-o-sleep</td>
<td>Communication</td>
<td>A p2p filesharing network consisting of records of all the mp3 files shared and transferred over 81 days.</td>
</tr>
<tr>
<td>KDD Cup 1999*</td>
<td>TCP traffic</td>
<td>Labeled network traffic data that is a version of the simulated network data made available by the 1988 DARPA Intrusion Detection Evaluation Program.</td>
</tr>
<tr>
<td>Facebook</td>
<td>Social</td>
<td>A list of Facebook wall posts, where a wall posts between two users creates a timestamped edge.</td>
</tr>
<tr>
<td>Congress spon-</td>
<td>Co-sponsorship</td>
<td>The co-sponsorship networks of 280,000 pieces of legislation proposed in the U.S. House and Senate from 1973 to 2004.</td>
</tr>
</tbody>
</table>
Table A.5 Methods with open source code, the types of graphs they work on, the language they are written in, and the output format; *These methods operate on the community belongingness matrices of graphs, therefore they are applicable to any graph that can have communities extracted.

<table>
<thead>
<tr>
<th>Code</th>
<th>Directed</th>
<th>Undirected</th>
<th>Weighted</th>
<th>Unweighted</th>
<th>Plain</th>
<th>Attributed</th>
<th>Parameterized</th>
<th>Streaming</th>
<th>Language</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayesian [82]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MATLAB</td>
<td>[0, 1] per time step and [0, 1] per edge per time step</td>
<td></td>
</tr>
<tr>
<td>DeltaCon [58]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>MATLAB</td>
<td>[0, 1] per time step</td>
<td></td>
</tr>
<tr>
<td>NetSpot [119]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>Java</td>
<td>Subgraphs with time intervals</td>
<td></td>
</tr>
<tr>
<td>CTOOutlier [75]*</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Java</td>
<td>[0, 1] per vertex for all time steps</td>
<td></td>
</tr>
<tr>
<td>ECOOutlier [76]*</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Java</td>
<td>[0, 1] per vertex per community per time step</td>
<td></td>
</tr>
<tr>
<td>ParCube [131]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MATLAB</td>
<td>Tensor factors for all time steps</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX

B

BITCOIN SUPPLEMENTARY INFO

Exchanges Used
Figure B.1 Frequency of exchanges in STBs.
Figure B.2 Relationship between the number of vertices owned by each exchange and the frequency of that exchange in STBs.
**Features Used**
An address' feature matrix is composed of the following features, extracted from each day of the data and then aggregated. Features prefixed with "*" are those removed in the second experiment, where exchange label based features are removed.

1. `total_bitcoin_received` – How much BTC the address received from transaction outputs over the full time window.
2. `total_bitcoin_spent` – How much BTC the address spent as transaction inputs over the full time window.
3. `bitcoin_balance` – Total bitcoin received minus total bitcoin spent.
4. `num_predecessors` – How many unique addresses have been an input to transactions where this address was an output.
5. `num_transaction_outputs` – How many times this address has been used in a transaction output.
6. `num_successors` – How many unique addresses have been an output in transactions where this address was an input.
7. `num_transaction_inputs` – How many times this address has been used as a transaction input.
8. `num_siblings` – How many unique addresses have been co-inputs or co-outputs with this address.
9. `num_predecessor_exchanges` – How many unique exchange addresses have been an input to transactions where this address was an output.
10. `num_successor_exchanges` – How many unique exchange addresses have been an output in transactions where this address was an input.
11. `num_sibling_exchanges` – How many unique exchange addresses have been co-inputs or co-outputs with this address.
12. `num_gamma_patterns` – How many times this address is part of a $\gamma$ pattern.
13. `num_beta_patterns` – How many times this address is part of a $\beta$ pattern.
14. `num_L1_patterns` – How many times this address is part of an $L_1$ pattern.
15. \textit{num\_L2\_patterns} – How many times this address is part of an \(L_2\) pattern.

16. \textit{num\_alpha\_patterns} – How many times this address is part of a \(\alpha\) pattern.

17. \textit{num\_alphaprime\_patterns} – How many times this address is part of a \(\alpha'\) pattern.

18. \textit{reciprocity} – How many of this addresses successors are also predecessors.

19. \textit{anti\_reciprocity} – How many of this addresses predecessors are also successors.

We focused on local features that are fast to compute. Examples of more expensive but potentially very useful features are explained in [121], e.g. peeling chains or coinbase transactions.