ABSTRACT
A massive amount of data generated today on platforms such as social networks, telecommunication networks, and the internet in general can be represented as graph streams. Activity in a network’s underlying graph generates a sequence of edges in the form of a stream; for example, a social network may generate a graph stream based on the interactions (edges) between different users (nodes) over time. While many graph mining algorithms have already been developed for analyzing relatively small graphs, graphs that begin to approach the size of real-world networks stress the limitations of such methods due to their dynamic nature and the substantial number of nodes and connections involved.

In this paper we present GRAPHZIP, a scalable method for mining interesting patterns in graph streams. GRAPHZIP is inspired by the Lempel-Ziv (LZ) class of compression algorithms, and uses a novel dictionary-based compression approach in conjunction with the minimum description length principle to discover maximally-compressing patterns in a graph stream. We experimentally show that GRAPHZIP is able to retrieve complex and insightful patterns from large real-world graphs and artificially-generated graphs with ground truth patterns. Additionally, our results demonstrate that GRAPHZIP is both highly efficient and highly effective compared to existing state-of-the-art methods for mining graph streams.

KEYWORDS
Graph mining; Streaming data; Compression

1 INTRODUCTION
Graphs are used to represent data across a wide spectrum of areas, from computational chemistry to social network analysis. Graph mining is an active area of research, and there are numerous methods for mining smaller graphs (several thousand edges), but many of these systems are unable to scale to real-world graphs (e.g., social networks) with millions or even billions of edges. Conventional graph mining algorithms assume a complete static graph as input, however many real-world graphs are often too large to hold in main memory. Additionally, many real-world graphs of interest are dynamic and actively growing - Facebook, for example, records over 300 new users per minute and has a social graph with more than 400 billion edges [10]. While it is possible to utilize conventional graph mining systems on dynamic graphs by processing static ‘snapshots’ of the graph at various points in time, in many cases the underlying data the graph represents changes at a rate so fast that attempting to analyze the data using such methods is futile.

In cases where the graph in question is inherently dynamic, we can instead treat the graph as a sequential stream of edges representing continuous updates to the graph’s overall structure. For example, given a graph modeling friendships (edges) between users (nodes) in a social network, we can consider all new or updated relationships during a set time interval (e.g., 1 hour) a set of edges from time \( t_i \) to \( t_{i+1} \). The graph mining system then processes the sequential edge sets at every interval, as opposed to attempting to read the entire graph at once. Processing large graphs in a streaming fashion drastically reduces the system’s memory requirements (since only small portions of the graph are seen at a time) and enables processing of large, dynamic real-world datasets. However, deploying a streaming model for real-time data analysis also imposes strict constraints: the system has a limited time window to process each set of edges, and edges can only be viewed once before they are replaced in memory by those in the next set.

Many graph mining algorithms aim to identify interesting patterns within an input graph. Various algorithms use different metrics to quantify how ‘interesting’ a pattern is: frequent subgraph mining (FSM) focuses on finding all subgraphs that appear in the graph over a certain frequency threshold, whereas problems such as counting motifs or finding maximal cliques in a graph (formalized in [26] and [8], respectively) focus on discovering subgraphs with a specific structure.

This paper relies on a novel approach to identify interesting patterns in a graph - namely, finding a set of substructures that best compress the graph. More precisely, we compress a graph using a pattern (subgraph) \( G \) by replacing all instances of \( G \) in the graph with a new node \( p \) representing \( G \) (see figure 1). The reduction in size of the overall graph is a measure of the compression afforded by pattern \( G \), and we search for patterns that compress the graph to the maximal extent. The same concept is found in certain types of data compression (e.g., LZ78 [42], ZIP) where the compression method looks for recurring patterns or sequences in the data stream, builds a dictionary representing the recurring patterns with shorter binary codes, and then stores the compressed data using only the binary codes and the dictionary. In the context of graphs, a byproduct of this process is that the pattern dictionary contains a set of subgraphs that compress well, and therefore represents an alternative approach for finding interesting (highly-compressing) patterns in a graph stream.

We propose a dictionary-based compression method for graph-based knowledge discovery: GRAPHZIP. Our approach is designed to efficiently mine graph streams and uncover interesting patterns by finding maximally-compressing substructures. Specifically, our main contributions are as follows:
(1) We propose a new graph mining paradigm based on the LZ class of compression algorithms.
(2) Based on this paradigm, we introduce a new graph mining algorithm, GRAPHZIP, for efficiently processing massive amounts of data from graph streams.
(3) We demonstrate the effectiveness and scalability of our method using a variety of openly available synthetic and real-world datasets.

In our experiments, we demonstrate that our approach is able to retrieve both complex and insightful patterns from large real-world graphs by utilizing graph streams. In addition, we show that our approach is able to successfully mine a large class of varied substructures from artificially-generated graphs with ground truth patterns. When we compare GRAPHZIP’s performance with that of several other state-of-the-art graph mining methods, we find that GRAPHZIP consistently outperforms state-of-the-art methods on a variety of real-world datasets.

The GRAPHZIP system, including all related code and data used for this paper, are available for download online.

2 RELATED WORK

For the purposes of this paper, we classify previous work into two general categories: streaming and non-streaming.

2.1 Non-streaming

Non-streaming graph mining algorithms take as input either a single graph (single graph mining), or multiple smaller graphs (transactional mining).

Transactional mining. FSG [21] is an early approach to finding frequent subgraphs across a set of graphs, and adopts the Apriori algorithm for frequent itemset mining [4]. FSG works by joining two frequent subgraphs to construct candidate subgraphs, then checking the frequency of the new candidates in the graph. GSPAN [37] uses a ‘grow-and-store’ approach that extends saved subgraphs to form new ones, an improvement over FSG’s prohibitively expensive join operation. MARGIN [33] prunes the search space to find maximal subgraphs only, a narrower and thus easier problem than FSM. CloseGraph [38] is another method that reduces the problem space by mining only closed frequent subgraphs - subgraphs that have strictly smaller support than any existing supergraphs.

LEAP [36] and GraphSig [27] are two recent approaches for mining ‘significant subgraphs’ as measured by a probabilistic objective function. By mining a small set of statistically significant subgraphs as opposed to a complete set of frequent subgraphs, LEAP and GraphSig are able to avoid the problem of exponential search spaces generated by FSM miners with low frequency thresholds.

Single graph mining. SUBDUE [20] is an approximate algorithm based on the branch-and-bound search technique. SUBDUE, like GRAPHZIP, uses the Minimum Description Length (MDL) principle [29] to mine maximally-compressing patterns in the graph. However, unlike GRAPHZIP, SUBDUE returns a restrictively small number of patterns regardless of the size of the input graph [22]. SUBDUE has been improved in recent years [11–13], yet the fundamental limitations of the algorithm (in particular the branch-and-bound technique) remain the same. SEuS [18] is an approximate method that creates a compressed representation of the graph by collapsing vertices that share labels. SEuS however is only effective in cases where the input graph has a small number of unique subgraphs that occur with high frequency, as opposed to when the input graph has a large number of subgraphs that appear with lower frequency. SigGRAM [23] is a complete method for finding frequent connected subgraphs (complete methods are guaranteed to find all solutions that fit certain constraints such as the minimum frequency threshold, unlike their approximate counterparts). SigGRAM adopts a grow-and-store approach similar to G Span, but uses the expensive (NP-complete) Maximal Independent Set (MIS) metric for its frequency threshold, leading the system to be comparatively inefficient in practice. Additionally, like SEuS, SigGRAM suffers from a limited domain problem as it is designed specifically to mine sparse, undirected and labeled graphs only. GREW [22] is another approximate method for mining frequent connected subgraphs, and is similar to SUBDUE in that GREW only discovers a relatively small subset of solutions in the search space. GraMI [16] is a state-of-the-art complete method (with an approximate version AGRA MI) that has been shown to be highly-efficient for FSM on a single large graph. However, the size of the input graph is still limited since GraMI requires the entire graph to be held in main memory. Arabesque [32] is a recent distributed approach built on top of Apache Giraph [5] that can horizontally scale non-streaming algorithms (FSM, clique finding, motif counting, etc.) across multiple servers. However, horizontal scaling can be cost-prohibitive and is only capable of linearly scaling algorithms whose runtimes often grow exponentially with the size of the input graph. GERM [6] and the algorithm introduced by Wackersreuther et al. [35] can mine frequent subgraphs in dynamic graphs, however both methods require as input snapshots of the entire graph as opposed to incremental updates to the graph via graph streams.

2.2 Streaming

GRAPHSCOPE [30] is a parameter-free streaming method that, like GRAPHZIP and SUBDUE, is based on the MDL principle. GRAPHSCOPE encodes the graph stream with the objective of minimizing compression cost, in order to determine important change-points.
in the temporal data. Beyond change-point and community detection however, GRAPHSCOPE has limited use for other tasks, e.g. mining interesting subgraphs. Though the model itself is parameter-free, GRAPHSCOPE requires the dimensions of the graph (number of source and destination nodes) to be known a priori, and thus is unable to mine streams from dynamic graphs where unseen nodes in new edge streams. Braun et al. [7] proposed a novel data structure called DSMatrix for mining frequent patterns in dense graph streams, yet similar to GRAPHSCOPE their approach requires that the edges and nodes be known beforehand, limiting its real-world applications. Aggarwal et al. [1] introduced a probabilistic model for mining dense structural patterns in graph streams, however the approximation techniques used lead to the occurrence of both false positives and false negatives in the results set, reducing the method’s viability in many real-world settings. STREAMFSM [28], based on cSPAN, is a recently introduced method for frequent subgraph mining on graph streams, whose performance we compare directly with that of GRAPHZIP (see section §4). There also exist several systems targeted at more specific graph analysis tasks in the streaming setting: counting triangles [34], outlier [3] and hotspot [39] detection, and link prediction [40]. Summarization methods such as TCM [31], cSKETCH [41] and COUNT-MIN SKETCH [14] focus on constructing sketch synopses from large graph streams that can provide approximate answers to queries about the graph’s properties. For a detailed survey of state-of-the-art graph stream techniques, see [25] (for a more general overview of graph mining algorithms, see [2]).

GRAPHZIP can process an infinite stream of edges without requiring details about nodes or edges beforehand, and has no restrictions on the type of graph being streamed. While GRAPHZIP is designed specifically for the streaming setting, it draws from ideas such as grow-and-store and the MDL principle originally applied in non-streaming methods. In contrast to summarization methods, GRAPHZIP returns exact subgraphs extracted from the stream as opposed to approximate results. To the best of our knowledge, GRAPHZIP is the first graph mining algorithm for mining maximally-compressing subgraphs from a graph stream.

3 METHOD

3.1 Preliminaries

In this section, we review the fundamental graph theory needed to formulate our approach and formalize the definitions used in the rest of the paper. See Table 1 for symbol definitions.

Terminology. A graph $G$ is composed of a vertex set $V$ which contains all vertices (nodes) $v \in V$, and an edge set $E$ which contains all edges $e \in E$, each of which connects a source vertex to a target vertex. A subgraph $G'$ of $G$ is a graph composed of a subset of $G$'s vertices and edges. All vertices $v \in V$ and edges $e \in E$ have a unique index which refers to its internal location in the edge or vertex list (e.g., $v_1$ in $V = \{v_1, v_2, v_3\}$ has index 0, $v_2$ has index 1, etc.). In a vertex-labeled graph, there exists a one-to-one (i.e., unique) mapping from each vertex to a label, and in an edge-labeled graph the same mapping exists for the edge set. The value of labels within a graph is often domain-dependent: e.g., in a social network, vertex labels may correspond to a user type (e.g. ‘male’, ‘female’).

Table 1: Symbol definitions. Note that $\alpha$ and $\theta$ are the hyperparameters of the GRAPHZIP algorithm.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>Arbitrary graph</td>
</tr>
<tr>
<td>$V_G$</td>
<td>Vertex set of graph G</td>
</tr>
<tr>
<td>$E_G$</td>
<td>Edge set of graph G</td>
</tr>
<tr>
<td>$S$</td>
<td>Graph stream sequence</td>
</tr>
<tr>
<td>$S(i)$</td>
<td>Graph at time $i$ of stream $S$</td>
</tr>
<tr>
<td>$B$</td>
<td>A batch of edges from graph stream</td>
</tr>
<tr>
<td>$P$</td>
<td>Pattern dictionary</td>
</tr>
<tr>
<td>$p(i)$</td>
<td>Pattern (graph) $i$ in dictionary $P$</td>
</tr>
<tr>
<td>$V_{p(i)}$</td>
<td>Vertex (node) set of pattern $p(i)$</td>
</tr>
<tr>
<td>$E_{p(i)}$</td>
<td>Edge set of pattern $p(i)$</td>
</tr>
<tr>
<td>$C_{p(i)}$</td>
<td>Compression score of pattern $p(i)$</td>
</tr>
<tr>
<td>$F_{p(i)}$</td>
<td>Frequency (count) of pattern $p(i)$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Batch size</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Size threshold of $P$</td>
</tr>
<tr>
<td>$H(G)$</td>
<td>Compression scoring function</td>
</tr>
<tr>
<td>$I(G, G)$</td>
<td>Graph isomorphism function</td>
</tr>
<tr>
<td>$SI(G, G)$</td>
<td>Subgraph isomorphism function</td>
</tr>
</tbody>
</table>

while edge labels may correspond to different relationship types (e.g. ‘friend’, ‘family’, etc.).

Definition 3.1. Isomorphism: Two graphs $G_1$ and $G_2$ are isomorphic (denoted by $G_1 \cong G_2$) if there is a one-to-one mapping between the edges and vertices of $G_1$ and $G_2$. That is, each vertex $v$ in $G_1$ is mapped to a unique vertex $u$ in $G_2$, the two of which must share the same edges, i.e., be adjacent to the same vertices (if the graph is labeled, the vertices and edges must also share the same labels). $G_1 \cong G_2$ is equivalent to $G_1$ and $G_2$ sharing the same structure.

Definition 3.2. Subgraph isomorphism: Graph $G_1$ is considered a subgraph isomorphism of graph $G_2$ if it is an isomorphism of some subgraph $G_2'$ of $G_2$. The actual instance of $G_2'$ is called an embedding of $G_1$ in $G_2$. The subgraph isomorphism problem is a generalization of the graph isomorphism problem, and is known to be NP-complete [17] (unlike the graph isomorphism problem, the complexity of which is undetermined). Despite the problem’s complexity, many graph mining algorithms make heavy use of subgraph isomorphism checks for graph matching, and accordingly several optimizations have been made in the past decade which have significantly improved the efficiency of isomorphism (or subgraph isomorphism) checks in practice.

Definition 3.3. Graph stream: A graph stream $S$ can be represented as a chronological sequence of edges drawn from a graph.

$S = \{e_1, e_2, \ldots, e_n\}$

We can process the graph stream by segmenting the stream into distinct sets of edges, each set forming a single (possibly disconnected) graph stream object. In the case of a dynamic graph, updates to the graph can be viewed as new stream objects. In the rest of the paper we also refer to graph stream objects as batches, where batch size refers to the size of the stream object’s edge set (i.e., the number of edges in the batch).
Algorithm 1 GraphZip

1: Initialize P
2: while edges remain in stream do
3:   Construct graph B using α edges
4:   for each graph p in P do
5:     $E \leftarrow$ subgraph isomorphisms of p in B
6:     for each graph g in $E$ do
7:       $g' \leftarrow$ copy()
8:       for each e in $E_p$ do
9:         if e not in p then
10:            Extend $g'$ by new edge e
11:         else
12:            Add internal edge e to $g'$
13:       end if
14:     end for
15:     Mark each extended edge $e \in B$ as used
16:     if $g' \neq g$ then
17:       Add $g'$ to P
18:     end if
19: end for
20: end for
21: $R \leftarrow$ remaining unused edges in B
22: Add edges in R as single-edge patterns to P
23: end while
24: return P

3.2 Problem Formulation

Given a graph stream and a compression scoring function $H$, our objective function equates to maximizing the cumulative compression score of the entire pattern dictionary $P$:

$$f(G, H) = \arg\max_P \sum_i H(P^{(i)})$$  \hspace{1cm} (1)

The direct approach to solving for $f$ would require enumerating over all possible subgraphs of $G$, a computationally intractable task in most real-world scenarios since it would require storing the entirety of the graph stream, in addition to computing subgraph isomorphism checks over the entire graph. Therefore, we employ a heuristic algorithm to approximate such a solution.

3.3 The GraphZip Algorithm

GraphZip is a highly-scalable method for discovering interesting patterns in a massive graph. Inspired by dictionary-based file compression, GraphZip builds a dictionary of highly-compressing patterns from the given graph stream, which can be used directly or fed into a separate non-streaming algorithm (e.g., a maximal-clique finder). While GraphZip is designed specifically with graph streams in mind, the algorithm can be easily applied to smaller static graphs without modification: if GraphZip is given as input a single graph it will automatically partition it into batches of size $\alpha$ and process the graph as a stream. If the total number of edges in the graph (or number of edges remaining after $n$ iterations) is less than $\alpha$, GraphZip will process the graph as a single batch. This flexibility between input types allows us to compare GraphZip directly with non-streaming methods.

The general procedure of GraphZip is illustrated in figure 2. GraphZip is initialized with an empty dictionary $P$ with max size $\theta$ (provided by the user), which maps graphs to their frequency (count) and compression score. GraphZip collects arriving edges from the graph stream into batches of size $\alpha$ (also provided by the user), and runs the compress procedure on each batch $B$: if a pattern $p$ from the dictionary is embedded in $B$, GraphZip increments the frequency of the pattern in the dictionary and recomputes its compression score. Additionally, for each instance $i$ of pattern $p$ embedded in batch $B$, GraphZip extends $p$ by one edge length, tagging each of the edges from $B$ used to extend $p$. The new edges used to extend $p$ are the edges incident on $i$ that exist in batch $B$ but not in pattern $p$. GraphZip then adds the new extended pattern to $P$. After $P$ has been updated with all the extended patterns, the remaining untagged edges in $B$ are added as single-edge patterns to $P$. Our current reference implementation supports both undirected and directed edges, but not hyper-edges or self-loops. However, these limitations are implementation specific rather than inherent to the algorithm. Additionally, both representational variants can be converted to simple edges (a node and two edges).

If the dictionary exceeds size $2\theta$, the dictionary is sorted according to the compression scores and trimmed to $\theta$. A pattern’s compression score is computed as follows:

$$H(P^{(i)}) = (|E_{p^{(i)}}| - 1) \times (F_{p^{(i)}} - 1)$$  \hspace{1cm} (2)

This equates a pattern’s compressibility to a product of its size and frequency. We use $F_{p^{(i)}} - 1$ so that a pattern with a frequency of 1 has a compression score of 0, since a pattern that only appears
once affords no real compression to the overall graph. The same offset is applied to the pattern size in $|(E_{p(i)}| - 1)$ to reduce the weighting of single-edge patterns. Due to the fact that overlapping instances of a pattern in a batch are counted independently, it is possible that GraphZip will overestimate the compression value of large structures with many homomorphisms.

Note that the compression method used is intrinsically lossy, since GraphZip does not retain information on how each of the instances are connected to the rest of the graph. The main focus of our work is knowledge discovery in graph streams, so lossy compression is an appropriate trade-off for decreased complexity and increased performance. More work is necessary to make GraphZip lossless, for example in the case where it is necessary to fully reconstruct the original graph from the pattern dictionary.

See algorithm 1 for pseudo-code, and the online repository for a reference implementation.

3.4 Scalability
Speed and memory usage are critical properties of graph mining algorithms designed to mine large real-world graphs. A deployed graph mining system should be able to keep up with the flow of data in the dynamic graph setting, while summarizing a possibly infinite graph stream in memory. Memory usage in GraphZip is directly bounded by the maximum dictionary size ($\theta$), and is indirectly bounded by the batch size ($\alpha$), since the patterns within the dictionary cannot grow larger than the batch size (no subgraph isomorphisms of the pattern in the batch will exist). Both parameters $\theta$ and $\alpha$ can be modified to maximize performance given certain hardware limitations.

The bulk of the computation in the GraphZip algorithm happens while checking for embeddings of pattern $p$ in batch $B$ (find all subgraph isomorphisms of $p$ in $B$). Note that because each entry in the pattern dictionary is unique, none of the subgraph isomorphism checks are contingent on each other, and thus the loop can be naively parallelized across an arbitrary number of cores. This allows for large performance gains and means that an increase in dictionary size can be scaled linearly with an increase in cores. Even without parallelization of the subgraph isomorphism checks, GraphZip is still faster than other state-of-the-art graph mining systems (as described in section §4). See section §A for a formal runtime analysis.

4 EXPERIMENTAL EVALUATION
There are two main questions we focus on when evaluating our algorithm: does it generate objectively and subjectively good results (i.e. correct and interesting results, respectively), and does it generate them in a reasonable amount of time? To answer these questions we test GraphZip on an extensive suite of synthetic and real datasets ranging from a few thousand to several million nodes and edges (see table 3). Using these datasets, we benchmark GraphZip against three state-of-the-art, openly available graph mining systems: SUBDUE, GRAM, and STREAMFSM. Because there is no directly comparable method to GraphZip for mining maximally-compressing patterns in graph streams, we instead evaluate GraphZip against a non-streaming method for mining compressing patterns (SUBDUE), a non-streaming method for frequent subgraph mining (GRAM), and a streaming method for frequent subgraph mining (STREAMFSM). Highly-compressing patterns are often both large and frequent, so FSM methods serve as an appropriate comparison to GraphZip.

All experiments were run on a commodity server configured with an AMD Opteron 6348 processor (2.8 GHz) and 128GB of RAM.

4.1 Synthetic graphs
To test whether our algorithm outputs correct substructures, we utilize a tool called SUBGEN [11] to embed ground truth patterns with desired frequencies into an artificially generated graph. This allows us to test whether or not a graph mining system correctly surfaces known patterns we expect to be returned in the result set. Since both GraphZip and SUBDUE are designed to mine highly-compressing patterns from a graph, we embed large and frequent (i.e., highly-compressing) patterns in the graph, then record the number of ground truth patterns recovered. Given a set of embedded patterns $E$, and a set of patterns $R$ returned by our graph mining system, an embedded pattern $E^{(i)} \in E$ is considered matched if for some returned pattern $R^{(i)} \in R$, $R^{(i)} \approx E^{(i)}$. Thus, we calculate the fraction $a$ of embedded patterns recovered using the scoring metric

$$a = \frac{|\{E^{(i)} | E^{(i)} \in E \land R^{(i)} \in R \land E^{(i)} \approx R^{(i)}\}|}{|E|}$$

Which is equivalent to

$$\text{accuracy} = \frac{\text{matched patterns}}{\text{total patterns}}$$

We count a ground truth pattern as matched if it is found in GraphZip’s pattern dictionary after the final batch, or in SUBDUE’s case, if it is returned directly at the end of the program.

In addition to making the ground truth patterns highly-compressing, we also design the patterns to cover a wide class of fundamental graph patterns, including cliques, paths, stars and trees (see figure 3). This allows us to discern if a method has difficulty mining a certain type of structure (e.g., a poorly designed system may have trouble detecting cycles and therefore cliques). The naming scheme for each synthetic graph dataset is $N$-TYPE, where $N$ is the number of vertices in the embedded pattern and TYPE is a shorthand of the pattern type (e.g., 3-CLIQ is a graph with embedded 3-cliques). All synthetic graphs in table 2 are generated with 1000 nodes, 5000 edges, and 20%, 50% or 80% coverage (the percentage of the graph covered by instances of the pattern).
We use several large, real-world graph datasets to test the scalability of SUBDUE, the *^1 for runtime indicates the program was terminated after 1000 seconds (no accuracy shown).

**Table 2: GraphZip and SUBDUE runtime and accuracy on various synthetic graphs.** For runtime, lower is better. For SUBDUE, the *^1 for runtime indicates the program was terminated after 1000 seconds (no accuracy shown).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Cov. (%)</th>
<th>Runtime (sec.)</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GraphZip</td>
<td>SUBDUE</td>
</tr>
<tr>
<td>3-CLIQ</td>
<td>20</td>
<td>52.25</td>
<td>66.68</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>3.779</td>
<td>22.22</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>3.665</td>
<td>11.99</td>
</tr>
<tr>
<td>4-PATH</td>
<td>20</td>
<td>45.37</td>
<td>58.00</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>3.052</td>
<td>18.57</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>2.935</td>
<td>10.30</td>
</tr>
<tr>
<td>4-STAR</td>
<td>20</td>
<td>50.70</td>
<td>1000+</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>4.184</td>
<td>1000+</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>4.483</td>
<td>1000+</td>
</tr>
<tr>
<td>4-CLIQ</td>
<td>20</td>
<td>68.06</td>
<td>1000+</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>29.19</td>
<td>1000+</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>13.92</td>
<td>44.78</td>
</tr>
<tr>
<td>5-PATH</td>
<td>20</td>
<td>48.47</td>
<td>1000+</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>4.461</td>
<td>21.30</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>4.267</td>
<td>24.16</td>
</tr>
<tr>
<td>8-TREE</td>
<td>20</td>
<td>62.68</td>
<td>1000+</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>10.39</td>
<td>1000+</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>11.07</td>
<td>1000+</td>
</tr>
</tbody>
</table>

**Table 3: Details of real-world datasets used.** The average stream-rate (in edges per second) is calculated by dividing the total number of edges by the time span of the entire graph.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vertices</th>
<th>Edges</th>
<th>Labels</th>
<th>Stream-rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBER</td>
<td>3,774,218</td>
<td>16,512,783</td>
<td>418</td>
<td>1.4 × 10^-2</td>
</tr>
<tr>
<td>Higgs</td>
<td>304,691</td>
<td>563,069</td>
<td>4</td>
<td>9.3 × 10^-1</td>
</tr>
<tr>
<td>HetRec</td>
<td>108,451</td>
<td>241,897</td>
<td>5</td>
<td>7.8 × 10^-5</td>
</tr>
</tbody>
</table>

4.2 Comparison with SUBDUE

Table 2 shows the runtime and accuracy (eq. 3) for GraphZip and SUBDUE on the synthetic datasets. GraphZip is clearly faster than SUBDUE, taking an order of magnitude less runtime in most experiments. Decreasing the coverage across all pattern types increased the runtime for both systems. SUBDUE is unable to process half of the datasets (including all 4-STAR and 8-TREE experiments) in less than 1000 seconds, while GraphZip is able to process the same datasets in a fraction of the time with 99-100% accuracy in all cases.

Among the datasets SUBDUE is able to process, the greatest difference in accuracy lies in the clique datasets (3-CLIQ and 4-CLIQ), where SUBDUE misses approximately 10% of the embedded patterns. The GraphZip algorithm contains an explicit edge case to extend internal edges in a pattern with no new vertices, which enables GraphZip to capture cliques with high accuracy (see algorithm 1 and figure 2).

Our results indicate a stark difference in efficiency between GraphZip and SUBDUE: SUBDUE is significantly slower than GraphZip even on relatively small graphs with several thousand edges, and for graphs with certain classes of embedded patterns in them (stars and binary trees are particularly problematic). For this reason, when evaluating GraphZip with larger real-world graphs we focus on benchmarking against more scalable methods.

4.3 Real-world graphs

We use several large, real-world graph datasets to test the scalability of the GraphZip algorithm. See table 3 for further details.

4.4 Comparison with Grami

Despite being designed for mining highly-compressing patterns like GraphZip, SUBDUE has clear performance issues that restrict benchmarking it against GraphZip on large real-world graphs. Therefore we also compare GraphZip with Grami, a state-of-the-art graph mining system for frequent subgraph mining on large static graphs. In contrast to SUBDUE, Grami is efficient enough to process datasets with millions of edges, however Grami’s relative performance still allows us to motivate the need for graph mining algorithms designed explicitly to handle streaming data.

Grami takes as input a single graph file as opposed to a sequence of edges, so in order to simulate mining a dynamic graph with a non-streaming method we append the previous graph with the next set of edges at each iteration, initializing the graph with the first set of edges. Thus, each iteration represents a ‘snapshot’ of the growing graph. Since both methods are parameterized, we first tune Grami’s parameter values for each dataset and then run each algorithm on the full dataset.
minimum frequency threshold so that it returns a usable number of non-single edge patterns, then set GRAPHZIP’s parameters (batch and dictionary size) such that the pattern dictionary resembles the set of subgraphs returned by GraMi. On NBER (figure 4), we fix GraMi’s minimum frequency threshold to 1000 which returned a set of subgraphs with a maximum, minimum, and average size of 6, 1, and 1.47 respectively. Running GRAPHZIP with $\alpha = 5$ and $\theta = 50$ resulted in a pattern dictionary with a maximum, minimum, and average subgraph size of 5, 1, and 2.89 (figure 6 shows the dictionary distributions for both NBER and HetRec). Since the overall runtime of each model depends significantly on the configuration of the parameters, the main purpose of our comparison is to examine trends in the runtime and stream-rate of each model using settings where they return comparable sets of subgraphs.

Our results show that GRAPHZIP is clearly more scalable than GraMi when mining large graphs in the streaming setting. While processing NBER, GraMi’s runtime (figure 4b) grows exponentially, experiencing a large spike near iteration 300. Figure 4a (normalized by patents per month) demonstrates this clearly: GRAPHZIP maintains a constant stream-rate throughout, while GraMi’s stream-rate gradually slows until it sharply drops near the final updates. In fact, GRAPHZIP’s stream-rate shows a slight increase over time; one explanation is that as the captured patterns in $P$ become more complex, less isomorphism checks occur per batch.

Results on the HetRec dataset indicate similar trends, though to a more extreme degree. With HetRec, we use a minimum frequency threshold of 9,000 for GraMi and keep the previous settings for GRAPHZIP: setting the threshold to 1,000 causes GraMi’s stream-rate to slow to a relative crawl, and when using 10,000, GraMi is able to process the entire dataset but only returns two frequent subgraphs. While processing HetRec with the threshold set to 9,000, GraMi maintains a high stream-rate which trends upwards over time until the 93rd iteration, where the system freezes and is unable to make any progress despite being left running for multiple days (as indicated by the red ‘X’ on figures 5a and 5b).

### 4.5 Comparison with STREAMFSM

Since there are no algorithms for mining highly-compressing subgraphs from graph streams in the existing literature, we benchmark GRAPHZIP against STREAMFSM, a recently developed streaming algorithm for frequent subgraph mining. Subgraphs that compress well are often both frequent and large, so the tasks of mining highly-compressing and frequently-occurring subgraphs are closely related. The STREAMFSM reference implementation available online was unable to find any frequently occurring subgraphs with any large datasets other than the provided HetRec dataset (we hypothesize this is likely due to an implementation error), so we report results for STREAMFSM on the HetRec dataset only.

A reasonable amount of time in the streaming setting equates to processing time less than or equal (at the very most) to the streaming-rate of the data; if the system cannot process the stream at the speed it is being generated, then the system is much less applicable in the real-life setting. Our results indicate that GRAPHZIP is significantly more scalable than STREAMFSM: while STREAMFSM’s stream-rate experiences an initial speedup, it quickly and consistently deteriorates after iteration 25, drastically increasing the runtime per iteration. The severe increase in runtime occurs around the same iteration that GraMi freezes (see figure 5b). In contrast,
The Higgs’s dataset has time data in seconds for each interaction, so we are able to pre-process the dataset into graph files segmented by the hour. One benefit to using the Higgs dataset is to observe how large spikes in network traffic affect the stream-rate; the minimum, maximum, and average number of edges streamed per hour are 43, 45,861, and 3,352 respectively, with the peak number of tweets per hour coinciding with the official announcement of the discovery.

As we can see in figure 7b, GRAPHZIP’s stream-rate is unaffected by the large spike in network traffic (using the same model parameters as the previous experiments). After an initial slowdown (similar to HetRec), GRAPHZIP’s stream-rate converges on a constant stream-rate slightly faster than the maximum stream-rate the network reaches at the 80 hour mark, and much faster than the average stream-rate of the network (9.3 × 10^{-1} tweets per second).

Our results indicate that if GRAPHZIP had been deployed to monitor the graph stream in real-time, it would have been able to process each set of updates before the next set of updates arrived.

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced GRAPHZIP, a graph mining algorithm that utilizes a dictionary-based compression approach to mine highly-compressing subgraphs from a graph stream. We showed that GRAPHZIP is able to successfully mine artificially-generated graphs for maximally-compressing subgraphs from a graph stream. We showed that GRAPHZIP is able to handle both complex and insightful patterns from large real-world graphs at speeds much faster than the actual stream-rate, with performance exceeding that of openly available state-of-the-art non-streaming and streaming methods. Future work will focus on implementing the potential optimizations to the algorithm discussed in this paper, including approximation algorithms for (subgraph) isomorphism computations and naive parallelization.

A COMPLEXITY ANALYSIS

In this section we examine the time complexity of algorithm 1 in detail. We begin by analyzing the runtime per batch B from a stream of graph G. Given a batch B, for each pattern P[i] ∈ P we retrieve the set M of all embeddings (subgraph isomorphisms) of P[i] in B:

\[ O(|P| \times O(sl(P[i], B))) \]  

(5)

Then, for each embedding M(i) ∈ M, we (a) extend a copy of M(i) by one edge length in each direction and (b) add the new pattern to the dictionary:

\[ O(|P| \times O(sl(P[i], B)) + |M| \times ([V_{M(i)}] \times k + |P| \times O(l(M(i), P(i)))))) \]  

(6)

where k is the number of edges incident of u ∈ V_B shared with u ∈ M(i), and M(i) is the extended copy of M(i). Finally, we add the remaining edges in B as single-edge patterns (e) to P:

\[ O(|P| \times O(sl(P[i], B)) + |M| \times ([E_{M(i)}] \times k + |P| \times O(l(M(i), P(i)))) + |R| \times O(l(e, P[i]))) \]  

(7)
If we use the well-known VF2 algorithm to implement the subgraph and graph isomorphism functions, the time complexity for $S(I)$ and $L(I)$ simplify to $O(V^2)$ in the best case and $O(V^2 + V)$ in the worst case, where $V$ is the maximum number of vertices between the two graphs. Recall that our model is parameterized by $\theta$ and $\alpha$ (|P| and |E|, respectively), which directly bounds $k < |V_B| < 2\alpha$, $|M| < 2^\alpha$ (maximum number of possible subgraphs in B), $E_M^{|v|} < \alpha$, and $|R| < \alpha$. Substituting into the worst case using VF2, we get:

$$O(\theta \times ((2\alpha)! \times 2\alpha) + 2^\alpha \times (\alpha \times 2\alpha + \theta \times ((2\alpha)! \times 2\alpha))) + \theta \times 2^\alpha \times 2 = 2^\alpha \times (\alpha + \theta \times (2\alpha)! \times 2\alpha)) + \theta \times 2^\alpha \times 2 \tag{8}$$

Which simplifies to:

$$O(\theta \times (2\alpha)! \times \alpha + \theta \times (2^\alpha \times \alpha^2 + 2^\alpha \times \theta \times 2^\alpha \times \alpha + \alpha)) + \theta \times \alpha \tag{9}$$

And lastly:

$$O(\theta^2 \times 2^\alpha \times \alpha! \times \alpha) = O(\theta^2 \times \alpha!) \tag{10}$$

Since $\theta$ and $\alpha$ are provided as parameters (constants at run-time), eq. 10 theoretically collapses to constant time ($O(1)$), meaning that the GraphZIP algorithm scales linearly with the size of the overall graph. However, our complexity analysis illustrates an important trade-off in selecting the batch size and dictionary size, since too large a value for either parameter can exponentially increase the run-time per batch (increasing $\alpha$ is particularly costly).

Our theoretical complexity analysis reinforces our experimental findings: that subgraph isomorphism calculations (a known NP-complete problem) dwarf all other computations in the algorithm. A promising area of future work would be to incorporate existing approximation algorithms for the subgraph isomorphism problem to increase the efficiency of GraphZIP.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No.: 1460917 and 1646640.

REFERENCES