Towards Interpretable Graph Modeling with Vertex Replacement Grammars

Justus Hibshman Satyaki Sikdar Tim Weninger
Department of Computer Science & Engineering
University of Notre Dame
Notre Dame, IN, USA
{jhibshma,ssikdar,tweninge}@nd.edu

***Abstract*—An enormous amount of real-world data exists in the form of graphs. Oftentimes, interesting patterns that describe the complex dynamics of these graphs are captured in the form of frequently reoccurring substructures. Recent work at the intersection of formal language theory and graph theory has explored the use of graph grammars for graph modeling and pattern mining. However, existing formulations do not extract meaningful and easily interpretable patterns from the data. The present work addresses this limitation by extracting a special type of vertex replacement grammar, which we call a KT grammar, according to the Minimum Description Length (MDL) heuristic. In experiments on synthetic and real-world datasets, we show that KT-grammars can be efficiently extracted from a graph and that these grammars encode meaningful patterns that represent the dynamics of the real-world system.**

***Index Terms*—Graph mining, graph model, vertex replacement grammar**

I. Introduction

A common task in big data is to seek and find patterns hidden in enormous amounts of data. When the data takes the form of the graph, this goal is expressed as finding meaningful graphical substructures and other patterns that are hidden in the graph. Because of the prevalence of graph data and the importance of this task, dozens of graph models have been developed towards this goal [1]–[4]. Typically, these graph models make some assumptions about the shape or structure of the graph and encode the graph in interesting ways.

Some of the most widely used graph modeling techniques search for occurrences of specific structures, such as edges, triangles, various 4-node graphlets, and so on. Other techniques measure specific graph properties such as node centrality, degree, or measures of network robustness. What almost all of these methods have in common is the fact that they typically learn structures that are specified in advance [5].

We currently lack modeling tools that allow the graph itself to dictate which graph patterns are essential and then report these newfound properties in a meaningful and human-readable format.

A few approaches are closer to this ideal than most. Notable works such as gSpan [6], CloseGraph [7], and SUBDUE [8] search for arbitrary substructures in a graph that can be used to create a lossy compression of the graph. However, these existing tools do little to show how those structures connect to each other and the rest of the graph, and some of them have trouble scaling to large or even mid-sized graphs. Progress in graph entropy uses an information-theoretic approach to identify graph structures and is a promising direction but does not produce an interpretable model [9].

978-1-7281-0858-2/19/$31.00 ©2019 IEEE

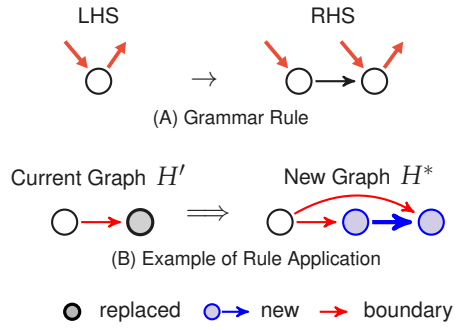

Fig. 1. (A) Example KT-grammar production rule with a left-hand side (LHS) and a right-hand side (RHS). The LHS is a single node with zero or more incoming and outgoing boundary edges (drawn in red). The RHS is a subgraph fragment, where each vertex has zero or more incoming and outgoing boundary edges. (B) During generation, a vertex from the graph is replaced by the RHS; incoming and outgoing boundary edges from the LHS are rewired to *all* of the incoming and outgoing boundary edges of the RHS respectively.

Renewed interest in graph grammars provides a promising route towards the goal of building a non-parametric, interpretable graph model. Previous work has investigated the relationship between graph mining and formal language theory by extracting Vertex Replacement Grammars (VRGs) [10] and (Hyper)edge Replacement Grammars (HRGs) [3], [11]. Unfortunately, the composition of grammar rules in HRGs, and some VRGs are known to produce clunky patterns that are difficult to interpret.

The present work uses the graph grammar introduced by Kemp and Tennenbaum (KT), which originally included a Bayesian graph model that could learn natural relationships between items in tiny datasets [12]. Generally speaking, KT-grammars, as we call them, are based on prior work in vertex replacement grammars, which contain graphical rewriting rules that can match and replace graph fragments similar to how a context-free string grammar rewrites characters in a string [13]. These graph fragments represent a succinct

description of the building blocks of the network, and the rewiring rules of the grammar describe the instructions about how the graph is pieced together.

KT-grammars, which are a specific type of VRG, are used to model graph structures and can even generate new graphs. A KT-grammar rule replaces a single *vertex* with a subgraph fragment as shown in Fig 1. KT-grammars are easy to use and easy to interpret, but their current use requires human modelers to craft these grammars by hand, which is time consuming and introduces human biases into the model. The rule inference system developed by Kemp and Tennenbaum has shown some promise in determining which rules best match data, but this system is limited to datasets of only a few dozen items [12]. We desire an automatic, scalable, and interpretable rule extraction algorithm that compactly models the structures found in the graph.

To that end, the present work describes BUGGE: a **B**ottom-**u**p **G**raph **G**rammar **E**xtractor (pronounced: "buggie"), which extracts interpretable KT-grammars from large real-world graphs. We show that the KT-grammar and BUGGE can correctly capture the known generative process of synthetic graphs. Upon their success in synthetic graphs, we employ BUGGE to find hidden structures in real-world graphs and report the findings.

II. Preliminaries

Before we describe BUGGE in detail, we first give some important background information. The BUGGE algorithm can take, as input, any graph $H = (V, E)$, which can be labeled, weighted, multi-edged, or directed. However, for simplicity, our implementation and the examples presented in this paper focus on simple, directed graphs with no edge weights or labels. Note that we use "vertex" and "node" interchangeably.

A. Vertex Replacement Grammars

A vertex replacement grammar is a context-free graph grammar consisting of a set of "production rules." These production rules (or simply "rules") prescribe a way to replace a single vertex in the graph with a subgraph fragment. When a vertex replacement occurs, the orphaned edges adjacent to the deleted vertex needs to be rewired to the new subgraph fragment in some way. Various edge rewiring schemes have been developed, each with their advantages and disadvantages.

The KT-grammar. The vertex replacement grammar introduced by Kemp and Tennenbaum is a natural formalism for our purposes [12]. This formalism, which we call a KT-grammar, is succinct and easy to interpret, but it is also rigid and sometimes requires algorithmic tradeoffs.

Formally, a KT-Grammar G is defined as a set of rules $R \in G$. Let $R = (F, i, o, f)$ such that $F = (V_R, E_R)$ is a directed graph fragment with vertices $v \in V_R$ and edges $e \in E_R$, $i : V_R \mapsto \{0, 1\}$ and $o : V_R \mapsto \{0, 1\}$ are indicator functions that state whether a vertex has incoming (i) or outgoing (o) boundary edges. $f \in \mathbb{Z}^+$ is the rule's "frequency," a count of how many times that rule occurs.

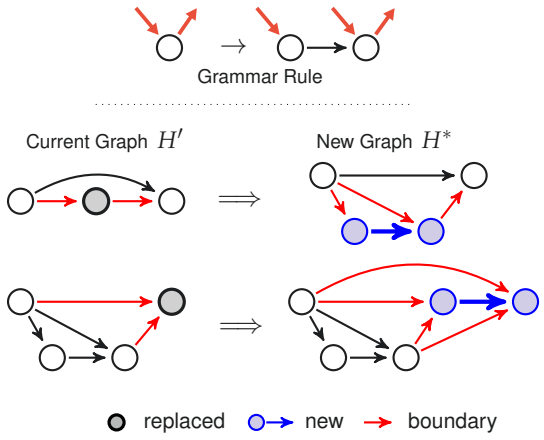

Fig. 2. A grammar rule repeatedly applied to grow a graph. At the top is the grammar rule; boundary edges, illustrated in red, indicate how the new RHS is rewired to the overall graph H. A red edge in a rule stands for connections to *all* a vertex's neighbors (0 or more). The next two rows show this rule applied to grey vertices in a H'. The expanded graph H^* is illustrated on the right where the rule's RHS is highlighted in blue.

Let $H = (V, E)$ be a directed graph upon which G is applied. Vertex replacement is defined as a transformation of H from a previous state H' to a new state H^* via $R \in G$. Let $v \in V$ be the vertex in H replaced by grammar rule $R = (F = (V_R, E_R), i, o, f)$. Then $H^* = (V^*, E^*)$, where the new vertices are:

$$V^* = (V' \setminus \{v\}) \bigcup V_R \tag{1}$$

and the new edges are:

$$\begin{aligned} E^* = \{(s,t) \mid\ & s, t \neq v \wedge ((s,t) \in E') \vee \\ & (s,t) \in E_R \vee \\ & (t \in V_R \wedge i(t) \wedge (s,v) \in E') \vee \\ & (s \in V_R \wedge o(s) \wedge (v,t) \in E')\} \end{aligned} \tag{2}$$

Simply put, whenever a rule replaces a node x, every node in R either gets all of x's boundary edges or none of them.

The example in Fig. 2 shows two additional applications of the rule from Fig. 1A. This single KT-grammar rule can be represented formally as $(F = (\{x, y\}, \{(x, y)\}), i, o, f)$ where $i(x) = 1$, $o(x) = 0$, $i(y) = 1$, $o(y) = 1$, and f is some positive integer.

Note that if a grammar rule has $i(v) = 0$ for all $v \in V$, then it cannot be used to replace a node with incoming edges. Likewise for outgoing boundary edges. Thus, the grammar rule's i and o functions implicitly define the left-hand side (LHS) of the rule. KT Grammar rules can have any number of nodes on the right-hand side (RHS).

B. Minimum Description Length Principle

The Minimum Description Length (MDL) principle asserts that the best representation of some data is the representation that uses the fewest bits. While this may be a questionable assertion philosophically, practically it is often a useful principle for big data mining and modeling. For example, gap-encodings can represent sparse matrices much more efficiently

than a direct "matrix" encoding because gap encodings better "match" the data [14].

The MDL principle may also be used in the following way: Given some data D, a set of models $\mathcal{M}$, and a particular encoding scheme E, the best model to encode D is the model $M \in \mathcal{M}$ that minimizes the combined cost of encoding D given M and the cost of encoding M (i.e. $E(D|M) + E(M)$).

In the present work, our data will be a graph, and our set of models will be a set of different grammar rules which our algorithm discovers. We will repeatedly, greedily select a grammar rule to compress the graph according to the MDL principle.

III. Extracting KT-Grammars

In this section, we describe BUGGE, and show that it can extract a succinct, meaningful KT-grammar that faithfully represents the graphical structures and properties of large graph data. We introduce BUGGE formally and then describe how it works using a running example.

A. BUGGE: the Bottom-Up Graph Grammar Extractor

Let $H = (V, E)$ be a directed graph with $V' \subseteq V$, and let $H(V')$ denote the subgraph in H induced by V'. We also introduce two size parameters $k_{\min}$ and $k_{\max}$ that bound the size of rule fragments. Let our grammar G start as an empty set.

At a high-level BUGGE extracts a vertex replacement grammar in the following way:

Step 1 Find all connected sets of nodes meeting the size constraints $k_{\min} \leq$ size $\leq k_{\max}$.

Step 2 For each connected set of nodes V', find the rules G' that could be used in reverse to contract V' into a single node. If no rules match exactly, find the rules which most closely match.

Step 3 Pick the single grammar rule R which is predicted to compress H the most.

Step 4 Extract an occurrence of $R \in G'$ from H by applying it in reverse. If R does not exactly match the nodes it collapses together, adjust the graph to make it fit (*i.e.*, add or delete edges). If R is not in our grammar G, add it to G. Increment the frequency of R in G.

Step 5 Update the sets of connected nodes and the associations of vertex sets to rules according to the new graph. Repeat Step 3 and Step 4 if there are still rules which can be extracted.

This principled approach extracts a vertex replacement grammar and can be applied to any kind of graph or grammar formalism. However, our goal of extracting a small, easily interpretable model is best satisfied by the KT-grammar formalism given earlier. So in the remainder of this section, we provide further details on how to extract a KT-grammar specifically.

Step 1: Enumerating Occurrences of Rules. When BUGGE first starts, it must enumerate rule occurrences for any connected set meeting the size constraints. Later, BUGGE only needs to perform updates to sets which might have been affected by the latest rule extraction (i.e., sets connected to nodes used in the latest extraction).

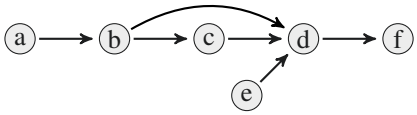

Fig. 3. Example directed graph. This graph will be used as a running example to show how BUGGE extracts KT-grammar rules.

Enumerating all connected sets of nodes up to a fixed size ($k_{\max}$ in our case) corresponds exactly to enumerating all connected, induced subgraphs up to some fixed size, which is solved using a technique called Reverse Search [15].

To show this process we introduce a running example using the graph shown in Fig. 3. With $k_{\min} = k_{\max} = 2$, there exists one connected subgraph per edge, 6 in total: {a,b}, {b,c}, {b,d}, {c,d}, {d,f}, and {e,d}. With $k_{\max} = 3$, we find 8 additional subgraphs: {a,b,c}, {a,b,d}, {b,c,d}, {b,d,e}, {b,d,f}, {c,d,e}, {c,d,f}, and {e,d,f}. The total number of sets for a graph tends to be exponential in $k_{\max}$.

Fortunately, the KT-grammar permits heuristics that allow the connected subgraph enumeration to be stopped early in many cases. If the enumerator just evaluated some set of nodes X of size $|X| < k_{\max}$ and we can infer that it is unlikely for any KT-Grammar rule including the nodes in X to be a "good" rule (more on this in Step 3), then the search can ignore more massive sets of nodes containing X. We use this "Enumeration Heuristic" in our experiments to speed up computation while retaining results of similar quality.

Step 2: Enumerating Rules that Apply to Subgraphs. Given the collection of connected sets, BUGGE finds the grammar rule(s) which best match each connected set.

For any given connected set, we consider every possible arrangement of boundary edges. In other words, for a given connected set V', we consider $R = (F, i, o, f)$ for every possible i and o, where F is the induced subgraph $H(V')$. For each rule (each possible i, o pair), we consider the minimum number of edge additions or deletions to H necessary to make V' correspond to an occurrence of R. This number of modifications is the "cost" of a rule occurrence. For a given V', we only store the rules with the lowest cost. Note that there are $2^{|V'|}$ possible i functions (and the same for o). Thus, this process is exponential in $k_{\max}$.

Figure 4 contains an example of finding matching rules for a connected subgraph induced by c and d. It illustrates two different rules that the subgraph could be edited to. Figure 4A shows that the boundary edge (b→c) does not match Rule 1, resulting in a cost of 1 for Rule 1. Figure 4B shows that the boundary edge (e→d) does not match Rule 2 resulting in a cost of 1 for Rule 2.

To make a decision in Step 3, BUGGE needs to aggregate information on all the occurrences of an individual rule. To do this, it assigns an id number to every rule. We begin with an

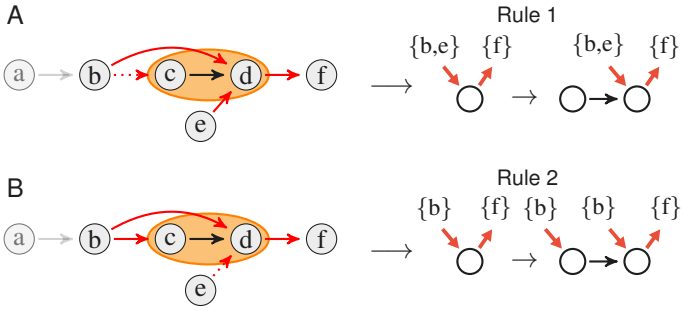

Fig. 4. Two options for extracting a KT-grammar rule from vertices {c, d} highlighted in orange. Red dotted edges indicate edge deletions necessary to make the graph match the rule on the right; edges deleted (or added) incur a cost to the model. Solid red edges are the preserved boundary edges. Labeled edges in the rules are for illustrative purposes only.

empty sequence of discovered rules, a "rule library", $L = \langle\rangle$. Every time an occurrence of a rule R is found in H, we check to see if R is isomorphic to a rule already in L. If not, we append it to L and give R a new id number. Otherwise, we give R the id of its match in L. To make this process more efficient, we maintain a count of how many times each rule has been discovered and adjust the order of rules in L to be in the order of discovery count, thereby increasing the likelihood that a newly discovered rule will match one of the first few rules in L.

Step 3: Finding the Rule with the Best Compression. At this point, each connected subgraph is matched with one or more possible rules, and each matching may have a non-zero cost associated. The next step is to decide which rule should be extracted from the graph. For this, we revisit the MDL principle introduced in the previous section. Simply put, we select the rule that we predict will minimize the overall description length of the graph given the grammar. See Appendix A for details on how we encode graphs via grammar rules and measure them in bits.

We predict the number of bits it will cost to use a rule n number of times as follows. Let R be a rule with multiple occurrences in graph H at various costs (number of edges to be added or deleted) $c_1, c_2, \ldots, c_m$, and let x_i denote the number of time rules R occurs in H at cost c_i.

We extract the cheapest occurrences of the rule first (*i.e.*, occurrences at cost c_1, followed by those at c_2, and so on.). Let $j(n)$ denote the highest cost index we would reach while extracting n rule occurrences and $X_{j(n)} = n - \sum_{i=0}^{j-1} x_i$ be the number of rules at cost c_j that we would select.

Let C_R denote the cost to encode the rule itself and give it an identification number. C_R will be zero if R has already been extracted and encoded. Let C_{ID} denote the cost to reference R's ID number. Due to our encoding scheme, we only need to reference this id once to perform a series of extractions using the rule. Let C_{node} be the cost to identify a single node in H (the node that the rule would be applied to). Lastly, let C_{edit} be the cost in bits to denote adding or deleting a node in H.

The predicted cost to encode n occurrences of R then becomes:

$$\text{COST}(n) = C_R + C_{\text{ID}} + nC_{\text{node}} + X_{j(n)}c_{j(n)}C_{\text{edit}} + \sum_{i=1}^{j(n)-1} x_i c_i C_{\text{edit}} \tag{3}$$

Recall that our MDL-based heuristic for selecting the most representative rule is to select the rule that lets us describe as much as possible with the fewest bits. Thus, what we really want to consider is not just the cost to encode some number n of grammar rule extractions but rather the cost in bits *per the number of nodes extracted*. We try to maximize the number of nodes per bit, which we refer to as the "Predicted Cost Ratio" (PCR). PCR for a rule is defined relative to the number of extractions that would yield the greatest ratio of nodes to bits.

Let n_i represent the number of nodes in H that would be extracted by a rule at a cost c_i. Thus, for a given n extractions with a rule, the predicted number of nodes to be extracted would be:

$$\text{NODES}(n) = \frac{X_{j(n)}}{x_{j(n)}} n_{j(n)} + \sum_{i=1}^{j(n)-1} n_i \tag{4}$$

The ideal predicted cost ratio (PCR) of nodes to bits for a rule R then becomes:

$$PCR_R = \max_n \frac{\text{NODES}(n)}{\text{COST}(n)} \tag{5}$$

If BUGGE were to compute this directly, it would require checking every possible n for each rule. Fortunately, it turns out that due to the "overhead" of the cost to encode and identify a rule, PCR will be maximized when *all* the occurrences at a given cost are extracted. Thus, the calculation of PCR can be simplified to:

$$PCR_R = \max_j \frac{\sum_{i=1}^{j} n_i}{C_R + C_{\text{ID}} + \sum_{i=1}^{j} x_i (C_{\text{node}} + c_i C_{\text{edit}})} \tag{6}$$

We choose the rule with the highest PCR as the best rule to extract. Although we compute the best number of occurrences to extract when determining the PCR of a rule, this value may change as soon as a single extraction is performed because the changes in the graph may remove other occurrences of R, causing R to have a worse PCR, or it causes some other rule R' to become cheaper or both. Hence, it should be stressed that this is a *Predicted* Cost Ratio.

Also, note that BUGGE assumes that the sets of nodes covered by a rule at different cost levels are disjoint. This is an idealized assumption and could lead to inaccuracies, although the PCR ratio appears to performs well in practice.

Returning to our running example, we find that the rule with the best PCR is Rule 1 from Fig. 4(A). Although it has an occurrence at cost 1 in Fig. 4(A), this rule occurs three other times: twice with a cost of 0 and once more with a cost of 2. The extra occurrences at cost 0 are what give Rule 1 the best PCR. Thus BUGGE selects one of the cheapest occurrences of Rule 1 to extract (either {a,b} or {e,d}).

Step 4: Extracting a Rule Occurrence. The rule extraction process "collapses" the induced subgraph by applying the rule in reverse. That is, instead of growing the graph by replacing a single vertex with a graph fragment as in Fig. 1 and 2, we reverse this process and extract a rule.

This processes is fairly straightforward. All the necessary edge additions or deletions were found when the rule occurrence was enumerated. Thus, BUGGE simply replaces the occurrence with a single node and rewires it according to the selected rule.

Returning to the running example, Fig. 5 illustrates an extraction of Rule 1 where it is calculated to have the lowest cost.

Step 5: Update and Repeat. An extraction changes the graph. So before we can iterate it is important that we update our record of rule occurrences.

To do this, we first determine which nodes have rule associations that may have changed due to the extraction in the previous step. Next, we delete registered rule occurrences involving any of the affected nodes. After the enumerations are updated, we repeat this process from Step 3.

The set of nodes which might be affected is upper-bounded by the set of nodes connected to the subgraph that was extracted. More specifically, it is the union of the following sets:

- The set containing the new "collapsed" node itself.
- Nodes in H for which an edge was deleted or added in the process matching the rule.
- Nodes in H which were connected by multiple in-edges or multiple out-edges to the collapsed subgraph, *i.e.*, boundary edges.

Again we return to the running example in Fig 5. After Rule 1 is extracted from the pair $\{e,d\}$, updates to rule occurrence enumeration occur for any set involving the newly created node g. Of particular interest, after extracting Rule 1, $\{c,g\}$ (formerly $\{c,d\}$) ceases to have Rule 1 as one of its cheapest rules, but $\{b,g\}$ then has an occurrence of Rule 1 at cost 1, and it is eventually selected. During its run on the example graph, BUGGE extracts the entire graph using Rule 1 multiple times, albeit with a non-zero cost (Fig. 5 C). Recall that a boundary edge in a KT grammar rule indicates that "all" (0 or more) edges get wired to that node. Some of the extractions in our running example have no (in or out) boundary edges.

Enumeration Heuristic. We found during testing that in practice, the rules with the cheapest edit costs are usually the rules with the best Predicted Compression Ratio (PCR). Thus, during the process of enumerating rules, it would only be important to enumerate the rules which have the lowest or near-lowest edit costs.

Consider a connected set X of $k < k_{\max}$ nodes with an edit cost of c. This means that there are c edges which must be added and/or deleted in order to extract X into a rule. The only way that adding another node to X could reduce the cost is if that node is one of the nodes that are connected to X via one of the edges which must be added or deleted. Furthermore,

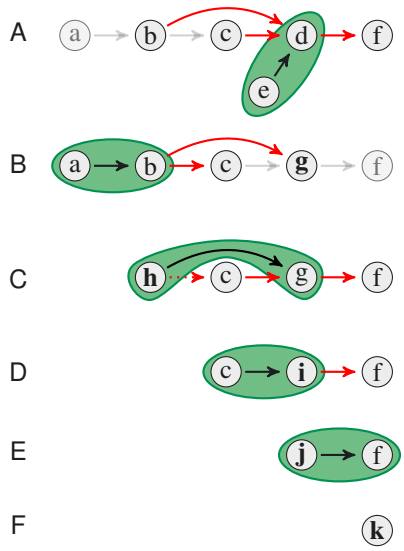

Fig. 5. BUGGE will repeatedly extract Rule 1 from Fig 4(A) thereby collapsing the entire example graph. Green areas highlight the nodes corresponding to a rule occurrence. Red arrows are boundary edges. The dotted red arrow in extraction (C) is an edge deletion. New vertices formed by the extraction of a rule (and the collapse of the relevant subgraph) in the previous step are labeled in bold. Note that the green highlighted nodes might lack in or out boundary edges. For example in (B), Rule 1 is extracted without cost from the subgraph a→b despite no incoming edges to b. This is compatible with the grammar rule because KT-grammars require boundary edges *that exist* to be rewired according to the rule.

this new node must not add any more edit cost. Thus, the chances of the cost decreasing as nodes are added are usually quite low.

BUGGE takes advantage of this observation. BUGGE stores the cost of the cheapest rule occurrence c_{best}; then, during rule occurrence enumeration, it updates this cost. If during enumeration, it finds that a set of nodes X has a cost which is "too far" from c_{best} then it doesn't bother to enumerate any connected sets of which X is a subset. We find that this provides a dramatic speedup.

More formally, we define a "shortcut parameter" s which tells BUGGE whether or not to enumerate larger sets. Specifically, we continue enumerating supersets of a set X with edit cost c ($X = k < k_{\max}$) if the following inequality holds:

$$c \leq c_{\text{best}} + \min\{1 + k_{\max} - k,\ s + \lceil \ln(k_{\max} - k) \rceil\} \quad (7)$$

In practice, we find that setting the shortcut parameter s to 1 tends to produce very similar results to running without a shortcut at all yet with a drastically reduced runtime (particularly noticeable when $k_{\max}$ is large). Larger values may increase runtime but produce better results. Sometimes we find that $s = 2$ will find interesting results that $s = 1$ will not; thus, s should be treated as a parameter that allows a potential tradeoff between results quality and runtime. Even with larger values of s however, the runtime is usually significantly reduced.

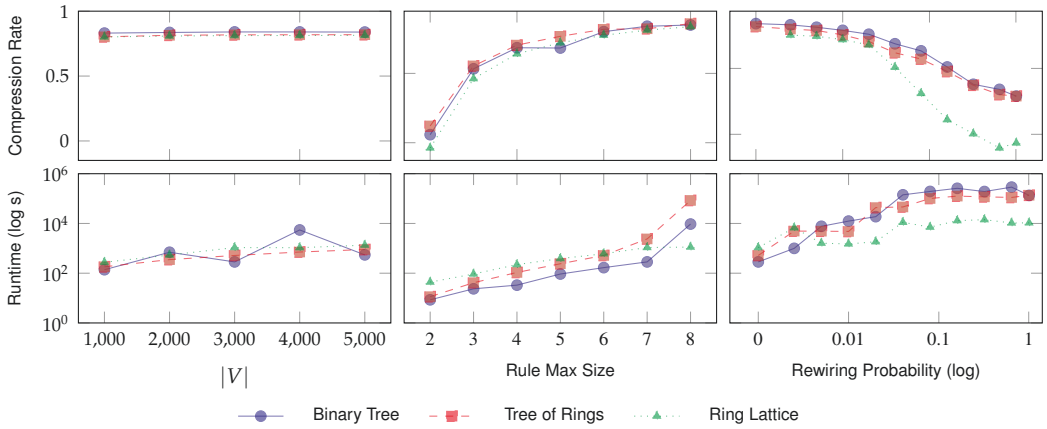

Fig. 6. BUGGE's Compression and Runtime results for synthetic graphs. We see how BUGGE responds as we vary different parameters: the size of the graph, the maximum allowed grammar rule size, and the amount of noise in the input (i.e. rewiring probability).

B. Related Work

MDL Approaches Of other well-known graph mining systems, our approach is most akin to SUBDUE [8], followed by VoG (Vocabulary of Graphs) [5]. Like our system, both SUBDUE and VoG use the MDL heuristic to select structures to extract.

SUBDUE uses a beam search to find structures. VoG searches for 6 preset structure types (cliques, stars, etc.) and maybe extended if the user wishes to implement support for other specific structures. Our system finds whatever structures are present in the graph up to a user-specified number of nodes. Thus, our approach lets the data "speak for itself" up to whatever computational costs the user is willing to allow.

Graph Grammar Approaches Other approaches extract a vertex replacement grammar using either a hierarchical clustering [10] or a tree decomposition of a graph [4] to select which nodes to collapse into a grammar rule. These approaches effectively try to form grammar rules from nodes that "go together." The Clustering-based Node Replacement Grammar (CNRG) provides a computational advantage over our approach. However, the choice of clustering algorithm adds a layer of indirection between the graph and the grammar which detracts from compressibility and interpretability.

The original work of Kemp and Tennenbaum did not extract grammars from a graph but instead tested if a dataset matched a particular grammar rule. This was particularly insightful because KT-grammars have two particular advantages.

First KT-grammars tend to be naturally interpretable when the intelligible structure can be found. Of particular noteworthiness, these rules can easily capture many of the structures which are most intuitive to the human mind: trees, rings, hierarchies, etc. For example, Table 7 shows some graphs along with grammar rules which generate them.

Second, KT-grammars are robust to error. For a given subgraph, there might not exist a rule that can create that particular subgraph. Usually, this happens when two nodes with external outgoing (or incoming) edges do not all point to the same nodes. At first glance, this might seem to be a weakness, but it enables an intuitive notion of the "cost" of applying a rule to a subgraph. This cost, defined as the number of edges in the graph that need to be added and/or deleted before the rule could apply, allows our algorithm to focus on the parts of the graph that most clearly correspond to interpretable structure, compressing those parts of the graph first.

Other Approaches Exponential Random Graph Models (ERGMs) are another type of graph model that learns a robust graph model from user-defined features of a graph [16]. Unfortunately, this model does not scale well and is prone to model degeneracy. Neural network graph models are of recent interest, but as is common with neural networks, these models do not provide the interpretability we desire. Additionally, some, such as GraphVAE [17] and GraphRNN [18] have limited scalability while others such as NetGAN [19] produce models many times larger than the original graph. Node embedding models like LINE [20], node2vec [21], VGAE [22], and others [23] represent individual nodes in the context of their local substructures for classification or prediction tasks and are also poorly suited to our objective.

IV. Methodology

In this section we present results of extensive experiments on real and synthetic datasets that compare compression, runtime, and model interpretability. We compare our results to several state-of-the-art graph summarization and grammar extraction methods including VoG [5], SUBDUE [8], and CNRG [10]. The source code for BUGGE, including experimental data and evaluation scripts, is available on GitHub[1].

Datasets. It is important that we consider both synthetic and real-world graphs in our evaluation. Synthetic graphs enable us to determine whether or not the grammar rules that we extract are interpretable, *i.e.*, since we know how we generate some synthetic graph, it's relatively easy to determine the goodness of the found graph substructures.

[1]https://github.com/SteveWillowby/ThreePartsTree

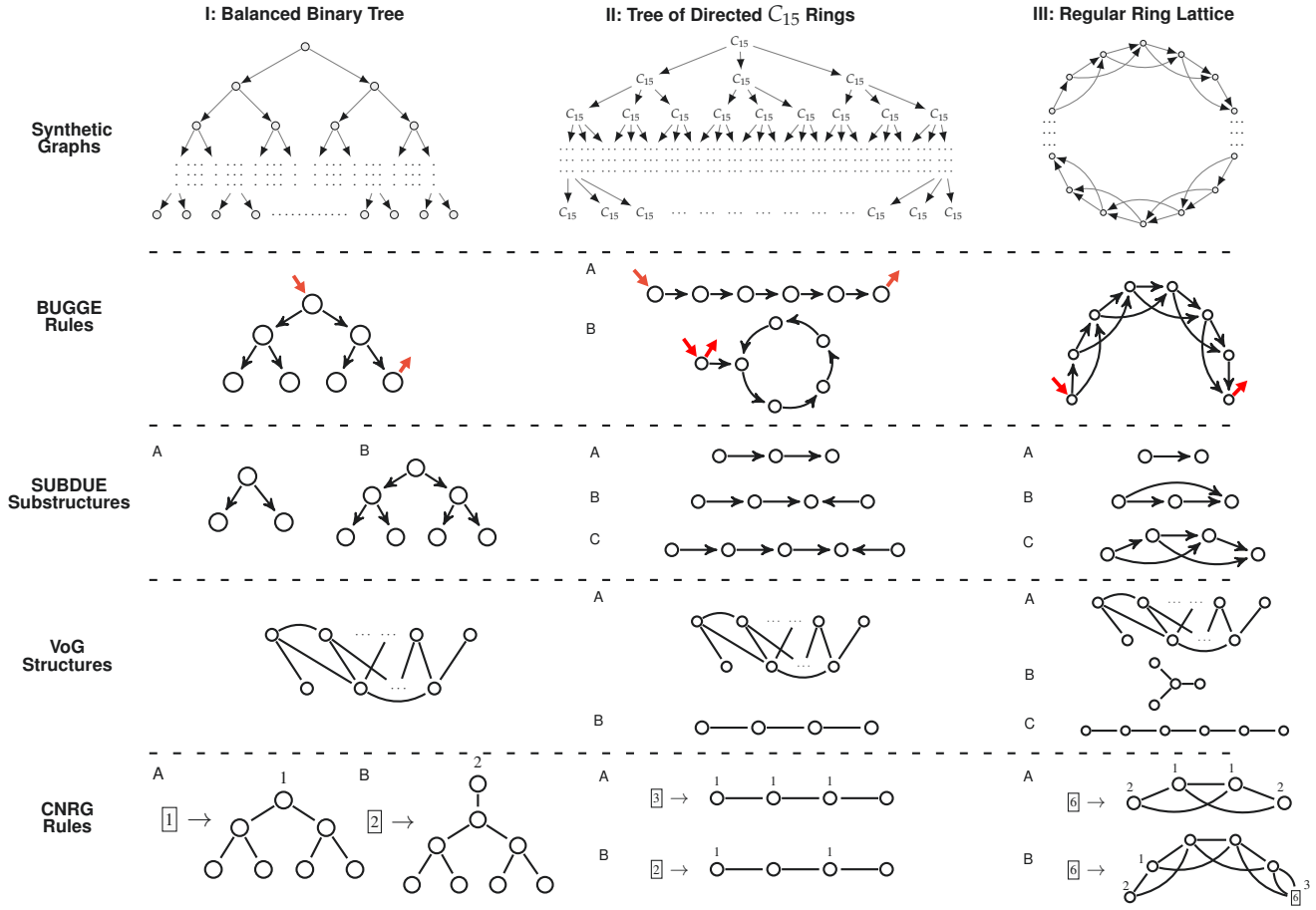

Fig. 7. Rules or substructures extracted by graph mining for three types of synthetic graphs. The rules extracted by BUGGE capture the known dynamics of the synthetic graph.

To that end, we generate three types of synthetic graphs: (1) a Binary Tree, (2) a Tree of Rings, which is an N-ary tree where each node is replaced with a ring of size k, and (3) a Ring Lattice, based on the Watts-Strogatz model of social networks. An ideal grammar extractor would describe these simple structures clearly.

In addition, we consider three real-world directed graphs from SNAP: Blogs (1,224 nodes, 19,025 edges), Protein-to-protein interaction network (1,706 nodes, 6,207 edges), and the DBLP citation graph (12,591 nodes, 49,743 edges).

Because runtime drastically increases with the maximum rule size, we use the enumeration heuristic in all of our tests. We find that it preserves the quality of results while improving runtime dramatically.

Synthetic Graph Results

First we test the runtime and compression ratio of BUGGE in various scenarios. Unless otherwise specified, graphs are generated with 3000 nodes; the N-ary tree has $n = 3$ with ring size $k = 15$; and the directed ring lattice graph has the degree set to 4.

Graph Size. Holding the rule size steady and the rewiring probability at 0%, we vary the number of nodes in the synthetic graph from 1000 to 5000. The results shown in Fig. 6(left) illustrates that runtime increases linearly in the number of nodes. This is what we expect given that the larger synthetic graphs just have more repetition of the same structure, so for a fixed $k_{\max}$ BUGGE just enumerates the same grammar rules more times.

The compression rate improves slightly on larger graphs. This matches our expectation because the overhead of defining more rules (bits increase) in larger graphs is dwarfed by the number of extractions with that rule (bits savings).

Rule Size. Holding the graph size and rewiring probability steady at 3000 and 0.0% respectively, we vary the maximum rule size from 2 to 8. Size-2 rules correspond to edges; size-3 rules can be one of the 5 directed 3-node graphlets. There are 34 different size-4 directed graphlets, and this number increases dramatically as the maximum allowed rule size increases [24], [25]. This increase in expressibility is certain to cause an increase in runtime. The results shown in Fig. 6(center) illustrates that the compression rate increases dramatically as the rule size increases (higher is better).

Model interpretability is explored in Fig. 7, which illustrates the most frequent rules extracted by BUGGE and the comparison methods where parameters are set empirically for each dataset. For example, in the synthetic binary tree graph Fig. 7(left), BUGGE extracts only two grammar rules,

one of which is (re-)used in 499 of the 501 total iterations. Thus, almost the entire graph can be represented with a single rule, which corresponds to replacing a node with a subtree. SUBDUE and CNRG extract reasonable rules from the binary tree, but VoG surprisingly extracts a nearly bipartite core.

For the tree of rings Fig. 7(center) illustrates two rules extracted by BUGGE that account for almost the entire graph (599 of the 601 extractions). First, a rule for a chain is used twice per ring to shrink the rings. Then a second rule takes one of the shrunken child-rings and wraps it entirely into its parent-ring. SUBDUE, VoG, and CNRG extract rules and substructures which are difficult to interpret.

For the ring lattice graph in Fig. 7(right), BUGGE extracts an intuitive rule that comprises 427 of the 430 total extractions. SUBDUE and CNRG produce reasonable results; however dozens of other CNRG rules are not illustrated here, and SUBDUE only associates its best rule with at most 68% of the graph. Again VoG produces a bipartite core.

These examples demonstrate how BUGGE can discern the nature of the original graph and common patterns within.

Random Rewriting Probability. To test the robustness of BUGGE to noise, we define a rewiring probability r. Holding the graph size at 3000, we vary r from 0 to 1. Before extraction, every edge is randomly re-assigned to a new pair of nodes with probability r. We design this process to ensure that the number of edges is preserved. This means that when r is 0 the synthetic graph remains the same and when r is 1 it becomes an Erdos Renyi graph.

Returning to Fig. 6(right), we observe that runtime increases significantly as the level of noise increases and compressibility drops. Interestingly, BUGGE manages to compress the graph with increasing levels of noise, thereby showing robustness; even when the rewiring probability is 1 (*i.e.*, entirely noise) BUGGE still manages to compress the two sparser random graphs, indicating that BUGGE can compress sparse noise.

Real-World Graph Results

The synthetic graph results show that BUGGE does indeed extract grammar rules that are meaningful. By inspecting the rules, we can discern certain aspects about how the graph is structured. Real-world graphs are less straightforward, but the goal remains the same: to extract meaningful rules that describe the underlying structure of the graph. Ideally, these rules will hint at the dynamics of the graph and shed light on the processes that govern these large, complex systems. We extracted grammars from 3 real-world graphs and inspected them to see what they tell us about the original graphs's structure.

Maayan Stelzl Protein-Protein Interaction (PPI) Graph. For the PPI network, almost all of the rules BUGGE finds have bidirected edges, suggesting that if protein A interacts with protein B, then the reverse is true. This is indeed the case; 95% of the connections in the original graph are bidirected.

The most frequently extracted rules are visualized in Fig. 8(top). By far the most frequent is a two-node rule

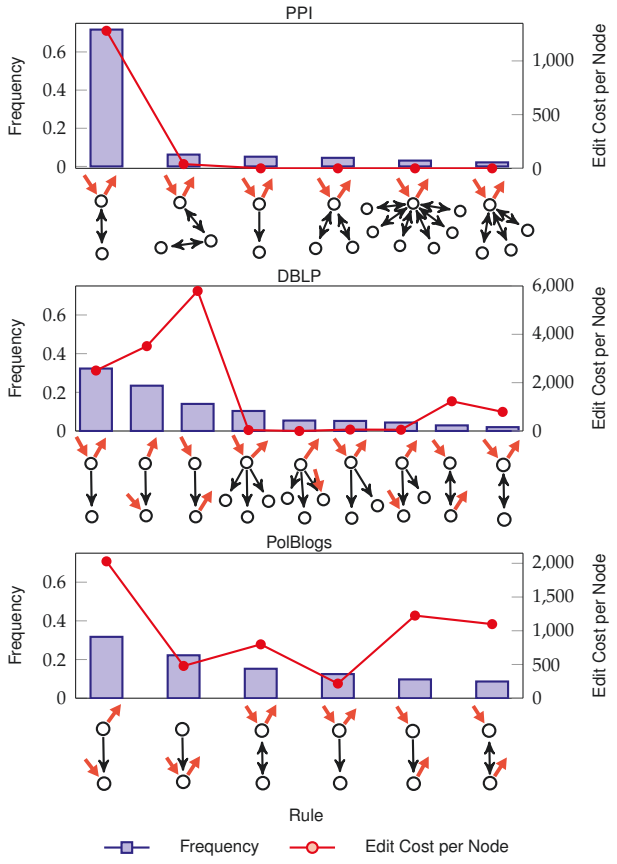

Fig. 8. Most frequently extracted rules from real-world graphs. The red line indicates the total edit cost (number of edges added or deleted) per node over the course of the extractions with that rule.

where one node has boundary edges, and the other does not. However, we find that in the course of 866 total extractions, 2561 edges were deleted (denoted by the red line). Thus, the node lacking edges in the rule typically had a few edges which were not held in common with its neighbor in the real graph. This suggests that the general structure of the graph is to have proteins with very few interactions (spokes) connect to proteins with very many (hubs). We especially see this "hub" trait in some of the other top rules illustrated in Fig. 8.

DBLP Article Citation Network. For the DBLP citation network, BUGGE extracts 9 rules which are used the most frequently. They are illustrated in Fig. 8(middle); many of which are similar to each other. As expected for a citation network, which should be a DAG, the most popular rules do not have bidirected edges.

We observed that in all of these rules, at most one node has outgoing boundary edges and at most one node has incoming. This means that for most pairs of connected nodes, it was cheapest for BUGGE to *delete* all but one node's in edges and *delete* all but one node's out edges. This, in turn, means that for most pairs of connected nodes, they had more distinct edges than edges in common. In terms of citations, this means a pair of articles connected by a citation are more likely to cite and be cited by different articles than by the same ones. This

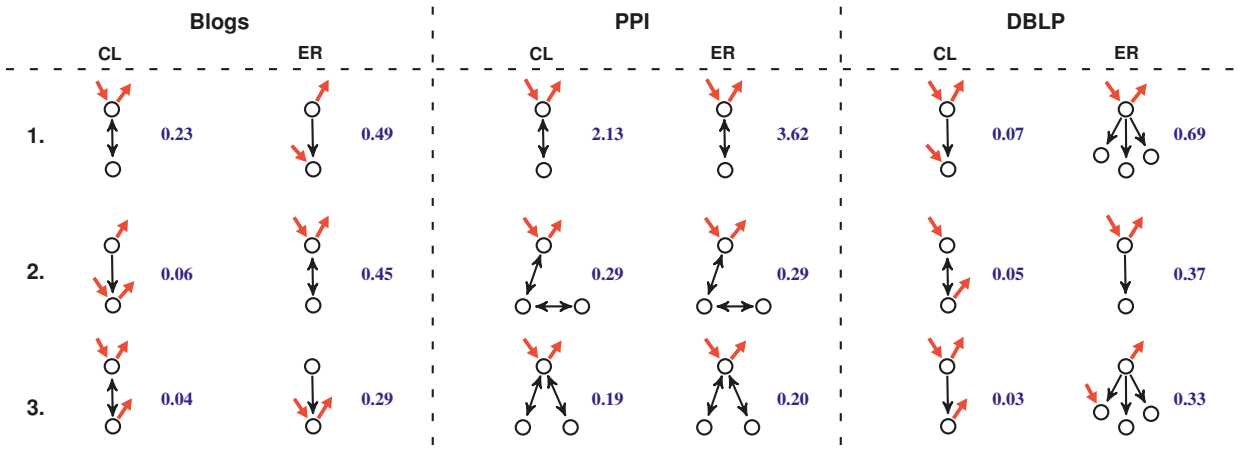

Fig. 9. Comparison of top 3 most "interesting" results when compared to Chung-Lu (CL) and Erdos-Renyi (ER) null models. BUGGE extracts KT-grammars that highlight certain dynamics of each dataset. Some patterns are well known, such as the bidirectional edges of PPI networks; others may require careful inspection and further study by domain experts.

level of expressibility is exactly what we seek; we, therefore, encourage domain experts in library sciences (or proteomics or the social Web) to investigate these findings further.

Moreno Blogs-Blogs Network. The Blogs network is another form of a citation network, but because multiple articles on the same blog count as the same node and two blogs can frequently cite one another, the Blogs network will not be nearly as DAG-like. Cycles and mutual citations should be much more common. We expected the blogs-to-blogs graph to have much less regular structure due to their complex social dynamics. However, we did obtain some of the same observations as in the DBLP citation network. In particular, that the shared citations between two blogs are fewer than the distinct citations.

Finding Interesting Rules

These rule probabilities give a good indication of the structure of the graph. However, it could be that some rules are just more likely than others, especially within graphs of the same degree distribution. So, it is important that we find the rules that are most *interesting* - not just most frequent. Defining what is "interesting" can be difficult; fortunately, null graph models are well suited for precisely this task.

For each real-world graph we create two null graph models: (1) an Erdos Renyi Random graph (ER) containing the same number of nodes and edges as the original graph, and (2) a random graph that matches the original graph's degree distribution using a directed version of Chung-Lu's Configuration model (CL) [26], [27]. We use BUGGE to extract a KT-grammar from the two null models for each real-world graph.

The extracted KT-grammars are a distribution of rules. So we can compare the graph models using KL-Divergence to determine how similar they are:

$$KL(p,q) = -\sum_{R \in \{G \cup G^{\emptyset}\}} p(R) \log \frac{q(R)}{p(R)}$$

where G is the KT-grammar extracted from the original graph, and $G^{\emptyset}$ is the KT-grammar extracted from the null model, either ER or CL; $p(R)$ and $q(R)$ are the probabilities that R appears in the grammar extracted from the original graph and the null model respectively. In some cases a rule may not appear in both graphs, so we perform Laplacian smoothing on these distributions to avoid errors caused by dividing by zero.

The KL divergence result itself is not particularly meaningful, however, the contribution of each rule R to the overall result represents the relative difference in their occurrence. Therefore, we rank each rule's contribution to the overall KL divergence and illustrate the top 3 rules in Fig. 9 for comparisons of real-world datasets against the null models.

Many aspects of our results could be commented on. We will highlight a few: The frequency of rules with bidirected edges in Figure 9 shows that neither the degree distribution nor the ER model capture these relationships. In the Blogs vs. CL comparison, we see that even though (as discussed earlier) most of the extracted rules do not have multiple out edges, they are more common in the original graph than the degree distribution alone would dictate. In the DBLP citation graph vs. ER, we see that BUGGE finds the original graph has much more tree-like/DAG-like rules.

Performance Comparisons

Finally, a direct comparison of the compression rates of BUGGE, SUBDUE, CNRG, and VoG is problematic. CNRG and SUBDUE are lossy models, while BUGGE and VoG are lossless models.

Likewise, direct runtime comparisons are also problematic. For example, the default settings for SUBDUE search for grammars of arbitrary size, which does not scale to even medium-sized graphs; so we set its max structure size to 8. Each algorithm is written in different programming languages using different graph libraries, etc. Runtimes in comparisons ranged from less than a minute on the smallest graphs to around 18 hours for BUGGE on the largest real-world graph.

V. Conclusions

The present work describes BUGGE: the Bottom-Up Graph Grammar Extractor, which extracts grammar rules that represent interpretable substructures from large graph data sets. Using synthetic data sets we explored the expressivity of these grammars and showed that they clearly articulated the specific dynamics that generated the synthetic data. On real-world data sets, we further explored the more frequent and most interesting (from an information-theoretic point of view) rules and found that they clearly represent meaningful substructures that may be useful to domain experts. This level of expressivity and interpretability is needed in many fields with large and complex graph data.

In future work, we intend to focus on extending these formalisms to cover temporal/evolving graphs, like the work done in synchronous HRGs [28] and temporal motifs [29]. It is also likely that the KT-grammars extracted here can be used to generate faithful null models of a graph.

Acknowledgements. This research is supported by a grant from the US National Science Foundation (#1652492). Thanks to the reviewers for their useful feedback. Lastly, thanks to Trenton Ford for his editing help.

References

[1] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield, "Efficient graphlet counting for large networks," in *ICDM*. IEEE, 2015, pp. 1–10.

[2] C. Seshadhri, T. G. Kolda, and A. Pinar, "Community structure and scale-free collections of erdős-rényi graphs," *Physical Review E*, vol. 85, no. 5, p. 056109, 2012.

[3] S. Aguiñaga, R. Palacios, D. Chiang, and T. Weninger, "Growing graphs from hyperedge replacement graph grammars," in *CIKM*. ACM, 2016, pp. 469–478.

[4] S. Aguinaga, D. Chiang, and T. Weninger, "Learning hyperedge replacement grammars for graph generation," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 41, pp. 625–638, 2019.

[5] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos, "Vog: Summarizing and understanding large graphs," in *SDM*. SIAM, 2014, pp. 91–99.

[6] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *ICDM*. IEEE, 2002, pp. 721–724.

[7] ——, "Closegraph: mining closed frequent graph patterns," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge Discovery and Data Mining*. ACM, 2003, pp. 286–295.

[8] L. B. Holder, D. J. Cook, S. Djoko *et al.*, "Substucture discovery in the subdue system." in *SIGKDD*, 1994, pp. 169–180.

[9] V. Gudkov, "Generalized entropies of complex and random networks," *Mathematical Foundations and Applications of Graph Entropy*, vol. 6, pp. 41–61, 2016.

[10] S. Sikdar, J. Hibshman, and T. Weninger, "Modeling graphs with vertex replacement grammars," in *ICDM*. IEEE, 2019.

[11] R. Reddy, S. Chandar, and B. Ravindran, "Edge replacement grammars: A formal language approach for generating graphs," in *SDM*. SIAM, 2019, pp. 351–359.

[12] C. Kemp and J. B. Tenenbaum, "The discovery of structural form," *PNAS*, vol. 105, no. 31, pp. 10 687–10 692, 2008.

[13] H. Ehrig, G. Rozenberg, and H.-J. rg Kreowski, *Handbook of graph grammars and computing by graph transformation*. World Scientific, 1999, vol. 3.

[14] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[15] D. Avis and K. Fukuda, "Reverse search for enumeration," *Discrete Applied Mathematics*, vol. 65, no. 1-3, pp. 21–46, 1996.

[16] G. Robins, P. Pattison, Y. Kalish, and D. Lusher, "An introduction to exponential random graph (p*) models for social networks," *Social Networks*, vol. 29, no. 2, pp. 173–191, 2007.

[17] M. Simonovsky and N. Komodakis, "Graphvae: Towards generation of small graphs using variational autoencoders," in *International Conference on Artificial Neural Networks*. Springer, 2018, pp. 412–422.

[18] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, "Graphrnn: Generating realistic graphs with deep auto-regressive models," *arXiv preprint arXiv:1802.08773*, 2018.

[19] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann, "Netgan: Generating graphs via random walks," *arXiv preprint arXiv:1803.00816*, 2018.

[20] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *WWW*, 2015, pp. 1067–1077.

[21] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *SIGKDD*. ACM, 2016, pp. 855–864.

[22] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.

[23] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowledge-Based Systems*, vol. 151, pp. 78–94, 2018.

[24] A. Sarajlić, N. Malod-Dognin, Ö. N. Yaveroğlu, and N. Pržulj, "Graphlet-based characterization of directed networks," *Scientific reports*, vol. 6, p. 35098, 2016.

[25] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[26] W. Aiello, F. Chung, and L. Lu, "A random graph model for massive graphs," in *STOC*. Acm, 2000, pp. 171–180.

[27] M. E. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distributions and their applications," *Physical review E*, vol. 64, no. 2, p. 026118, 2001.

[28] C. Pennycuff, S. Sikdar, C. Vajiac, D. Chiang, and T. Weninger, "Synchronous hyperedge replacement graph grammars," in *International Conference on Graph Transformation*. Springer, 2018, pp. 20–36.

[29] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *WSDM*. ACM, 2017, pp. 601–610.

Appendix

When compressing a graph, we use three encodings: A **graph encoding** (stores a minimalist adjacency list), a **grammar rule encoding**, and an **application encoding** (stores a sequence of applications of the grammar rules to a graph).

Graph Encoding. Given a directed graph $H = (V, E)$, the number of bits B_H it takes to encode G is:

$$B_H = (2\lceil \log_2 |V| \rceil - 1) + |V| + |E|(\lceil \log_2 |V| \rceil + 1)$$

Grammar Encoding. A grammar rule is basically a graph with additional boundary information. The total number of bits B_{R_k} to encode a grammar rule with k nodes is:

$$B_{R_k} = \lceil \log_2 |V| \rceil + k(\lceil \log_2 k \rceil + 2) + k(k-1) + 1$$

Application Encoding. An application encoding consists of a sequence of instructions for applying grammar rules. These instructions include an id number of the rule to apply, the id of the node to apply the rule to, and information concerning any edges which were added or deleted during the extraction process. The bits $B_{A_{km}}$ to record the application of a k-node rule with m edge approximations takes:

$$B_{A_{km}} = 2 + \lceil \log_2 |V| \rceil + m(\lceil \log_2 k \rceil + \lceil \log_2 |V| \rceil + 1) + \begin{cases} \lceil \log_2 |V| \rceil & \text{different rule used before} \\ 0 & \text{same rule used before} \end{cases}$$