Compressing Graphs by Grammars

Compressing Graphs by Grammars

Sebastian Maneth
School of Informatics
University of Edinburgh
Email: smaneth@inf.ed.ac.uk

Fabian Peternek
School of Informatics
University of Edinburgh
Email: f.peternek@ed.ac.uk

***Abstract*—We present a new graph compressor that detects repeating substructures and represents them by grammar rules. We show that for a large number of graphs the compressor obtains smaller representations than other approaches. For RDF graphs and version graphs it outperforms the best known previous methods. Specific queries such as reachability between two nodes, can be evaluated in linear time over the grammar, thus allowing speed-ups proportional to the compression ratio.**

I. Introduction

Large graphs have been gaining importance over the past years: be it RDF graphs and the semantic web or social networks such as Facebook. There is a plethora of recent research papers dealing with the analysis of large graphs, see e.g., [1]–[4]. Naturally, compression is an important technique for dealing with large graphs, cf. Fan [5]. It can be applied in many different ways. For instance, systems that use distributed processing (e.g., via Map/Reduce) need to repeatedly send large graphs over the network. Sending these graphs in a compressed form can have a huge impact on the performance of the whole system; see e.g., the Pegasus system [6], which applies off-the-shelf gzip compression to an appropriately permuted matrix representation of the graph [7]. Other applications are to use the compressed graph as in-memory representation, or, to build from it specialized indexes that support certain queries. An example for the first application is the compressed RDF graph representation by Álvarez-García et al. [8], an example for the second is the query-based compression by Fan et al. [9] which removes substructures from the graph that are not relevant for the supported class of queries.

Outside of the database community, compression of large network graphs has been studied already for more than a decade, possibly starting with the WebGraph framework by Boldi and Vigna [10]. Their methods proved very effective in compressing web graphs, but are less effective for other graphs, such as social graphs [11]. Furthermore, network graphs are commonly unlabeled and their methods are thus difficult to adapt to graphs with data, such as RDF graphs.

Grammar-based compression is an attractive formalism of compression. The idea is to represent data by a context-free grammar. Consider the string *ababab*. It can be represented by the grammar $\{S \to AAA, A \to ab\}$, whose size (sum of lengths of right-hand sides) is smaller than the given string. Remarkably, this simple formalism can be used to capture well-known compression schemes, such as Lempel-Ziv, see [12]. The attractiveness of grammar-based compression stems from its simplicity and mathematical elegance. It can, for instance, be used to explain one of the first grep-algorithms for compressed strings [13]. Besides grep there are many problems that can be solved in one pass through the grammar (see [14]), thus providing a *speed-up* that is proportional to the compression ratio. How can we find a smallest grammar for a given string? This problem is NP-complete [12]. Various approximation algorithms have been proposed, one of which is the RePair compression scheme invented by Larsson and Moffat [15]. It is a linear-time approximation algorithm that compresses well in practice. RePair was generalized to trees by Lohrey at al. [16]. Grammar-based compression typically produces two parts: a set of rules and an (often large) incompressible, remaining part.

We propose a generalization of RePair compression to graphs, to be precise, to directed edge-labeled hypergraphs. The idea of RePair is to repeatedly replace a most frequent digram (in a string, a digram is a pair of adjacent letters) by a new nonterminal, until no digram occurs more than once. In our setting, a digram consists of a pair of connected hyperedges. Let us consider an example: Figure 1a shows a grammar with initial graph S, and one nonterminal A, appearing three times in S. The rule for A generates a digram – two connected edges. We apply the rule by removing an A-edge, and inserting the digram, so that source and target nodes of the removed edge are merged with the source and target nodes of the digram. By applying the A-rule three times to the start graph, we obtain the terminal graph, which consists of three a- and b-edges as shown in Figure 1b. RePair compression intuitively does the reverse of the rule-application shown in Figure 1b. Starting with the full graph, it replaces edge pairs that occur more than once by nonterminal edges, and introduces corresponding rules. Thus it finds the digram consisting of an a- and a b-edge three times, and replaces each by an A-edge (introducing the A-rule). In general, the right-hand side of a rule can have several source and target nodes and we just speak of "external" nodes (e.g. the black nodes in Figure 1a). A rule with k external nodes is applied to a *hyperedge* with k incident nodes. It is well-known that restricting to rules (and start graph) with at most k nodes generates (hyper)graphs of tree-width at most k. The need to introduce hyperedges and right-hand sides with more than two external nodes can be outlined on a slight modification of the graph in Figure 1b, shown in Figure 1c. It has two additional c-edges. The most frequent digram is still the one with one a- and one b-edge. However now the center node must be external too. The extra two edges prohibit the center nodes removal. Thus an A-hyperedge with *three* incident nodes would replace the digram. Note that, in this example, no compression would be achieved, because the start graph still contains all original nodes, and because hyperedges are more expensive than ordinary ones.

Let us now discuss some of the challenges that come up when generalizing RePair to graphs. In each round of RePair

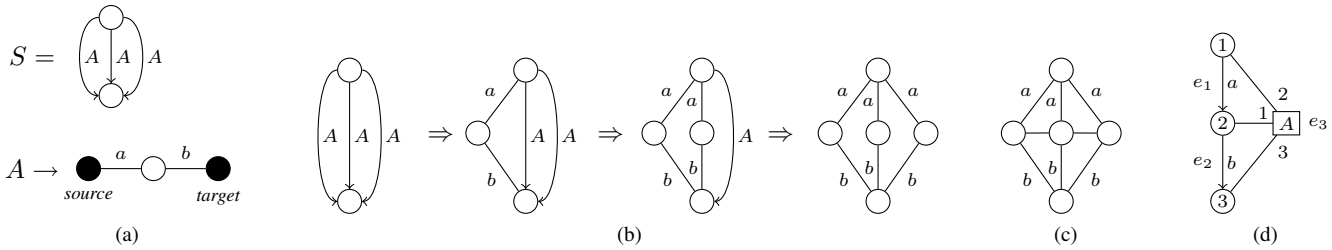

Fig. 1. Examples of a graph grammar (a), a full derivation of this grammar (b), a graph incompressible with gRePair (c), and a hypergraph (d).

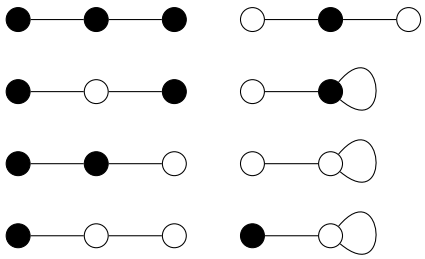

Fig. 2. Every possible digram using two unlabeled, undirected edges.

we need to find a digram with a maximal number of *non-overlapping* occurrences (i.e., occurrences that have no edge in common). In the classical string and tree cases of RePair, maximal non-overlapping occurrences of a digram can be found by simple greedy search. In the graph case, this is no longer possible. Determining such a subset is hard: it can be obtain from the full (overlapping) set of occurrences by *maximum matching*; the most efficient algorithms for the latter (such as "Blossom") require $O(|V|^2|E|)$ time, which is infeasible for large graphs. Furthermore, this is just the time required to find such a list for one digram. As seen in Figure 2, even for unlabeled, undirected graphs (without hyperedges) there are already eight different digrams to be considered. To address these challenges, we apply a greedy approximation. We follow some given fixed order ω of nodes. For every node u of degree k, we count the occurrences of d centered around u, in a way that only $O(k)$ possible digram occurrences are considered. The chosen node order can significantly affect the compression ratios. The main contributions of the paper are:

1) a generalization of RePair to graphs (gRePair),
2) an extensive experimental evaluation of an implementation of gRePair, and
3) a linear-time algorithm for (s,t)-reachability over grammar-compressed graphs.

The experimental results are that gRePair improves over state-of-the-art compressors for RDF graphs and for version graphs (unions of similar graphs). We found that one indicator of compression performance by our method is the number of equivalence classes of our particular node ordering (which is a generalization of the node degree ordering).

Related Work. Our grammar formalism is known as context-free hyperedge replacement (HR) grammars, see [17]. An approximation algorithm for finding a small HR grammar was considered already by Peshkin [18]. However, evaluation was only presented for rather small protein graphs. As far as we know, no other compressor for straight-line graph grammars has been considered. Claude and Navarro [19] apply string RePair on the adjacency list of a graph. This works well, but is outperformed by newer compression schemes such as k^2-trees. There are several approaches for web graph compression. The WebGraph framework [10] represents the adjacency list of a graph using several layers of encodings, while retaining the ability to answer out-neighborhood queries. A different encoding is proposed by Grabowski and Bieniecki [20], where contiguous blocks of the adjacency list are merged into a single ordered list. They then use gzip to compress this list, leading to the current state-of-the-art in compression/query trade-off, when only out-neighborhood queries are considered. The k^2-trees of Brisaboa et al. [21] compress the adjacency matrix by partitioning it into k^2 rectangles. If one of these includes only 0-values, it is represented by a 0-leaf in the tree, otherwise the rectangle is partitioned further. The method provides access to both, in- and out-neighborhood queries, and can be applied to any kind of binary relation. We use k^2-trees to represent the incompressible start graph of our grammars. The k^2-tree is used to compress RDF graphs in [8]. The k^2-tree-method was combined by Hernández and Navarro [22] with dense substructure detection, originally proposed by Buehrer and Chellapilla [23]. The method represents dense substructures as bicliques and replaces the edges between the two node sets with edges to a single "virtual node". More database oriented work is found for semistructured data. For example the XMill-compressor [24] groups XML-data such that a subsequent use of general-purpose compression (gzip) is more effective. Schema information can improve its effectiveness, but is not required. Deriving schema information from existing data can be seen as a form of lossy compression. DataGuides [25] are a way of doing just that for XML data.

II. Preliminaries

A *ranked alphabet* consists of an alphabet Σ together with a mapping $\mathsf{rank} : \Sigma \to \mathbb{N}\backslash\{0\}$ that assigns a rank to every symbol in Σ. For the rest of the paper, we assume that Σ is fixed, and of the form $\{1,\dots,n\}$ for some $n \in \mathbb{N}$. A *hypergraph over* Σ is a tuple $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$ where V is a set of nodes, E is a set of edges, $\mathsf{att} : E \to V^*$ is the attachment map, $\mathsf{lab} : E \to \Sigma$ is the label map, and $\mathsf{ext} \in V^*$ is a sequence of *external nodes*. We define the rank of an edge as $\mathsf{rank}(e) = |\mathsf{att}(e)|$ and require that $\mathsf{rank}(e) = \mathsf{rank}(\mathsf{lab}(e))$ for every edge in E. We add the following three restrictions on hypergraphs: (1) for all edges $e \in E$: $\mathsf{att}(e)$ contains no node twice, (2) ext contains no node twice, and (3) $V = \{1, 2, \dots, m\}$ for some m; these numbers are called *node IDs*. A hypergraph is *simple*, if (1) for all edges $e \in E$: $|\mathsf{att}(e)| = 2$ and (2) no two distinct edges $e_1, e_2 \in E$ exist such that $\mathsf{att}(e_1) = \mathsf{att}(e_2)$ and $\mathsf{lab}(e_1) = \mathsf{lab}(e_2)$. For a hypergraph $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$ we use

$V_g, E_g, \mathsf{att}_g, \mathsf{lab}_g$, and ext_g to refer to its components. We may omit the subscript if the hypergraph is clear from context. The rank of a hypergraph g is defined as $\mathsf{rank}(g) = |\mathsf{ext}_g|$. Nodes that are not external are called *internal*. We define the *node size* of g as $|g|_V = |V|$, the *edge size* as $|g|_E = |\{e \in E_g \mid \mathsf{rank}(e) \leq 2\}| + \sum_{e \in E_g, \mathsf{rank}(e) > 2} \mathsf{rank}(e)$, and the *total size* $|g| = |g|_V + |g|_E$. We denote the set of all hypergraphs over Σ by $\mathsf{HGR}(\Sigma)$. For sequences $w = x_1 x_2 \cdots x_n$ we write $x_i \in w$ to express that x_i is part of the sequence w. We also assume that the node IDs may represent arbitrary data values. Let D be a set of data values, then for every hypergraph, there is a mapping $\varphi : V \to D$ assigning values to nodes. We provide an example in Figure 1d. Formally, the pictured graph is $V = \{1, 2, 3\}$, $E = \{e_1, e_2, e_3\}$, $\mathsf{att} = \{e_1 \mapsto 1 \cdot 2, e_2 \mapsto 2 \cdot 3, e_3 \mapsto 2 \cdot 1 \cdot 3\}$, $\mathsf{lab} = \{e_1 \mapsto a, e_2 \mapsto b, e_3 \mapsto A\}$, and $\mathsf{ext} = \varepsilon$. Note that we omit the indices describing the order in which the nodes are attached to a hyperedge in the following. Instead, we use colors to indicate this order.

The next definition is a variant of context-free hyperedge replacement grammars, see, e.g. [17].

Definition 1. A *hyperedge replacement grammar (HR grammar) over* Σ is a tuple $G = (N, P, S)$, where N is a ranked alphabet of *nonterminals* with $N \cap \Sigma = \emptyset$, $P \subset N \times \mathsf{HGR}(\Sigma \cup N)$ is the set of *rules* such that $\mathsf{rank}(A) = \mathsf{rank}(g)$ for every $(A, g) \in P$, and $S \in \mathsf{HGR}(\Sigma \cup N)$ is the *start graph*.

In the literature (such as [17]), our restrictions $(1) - (3)$ on hypergraphs (see above) are not present. It is not difficult to show that these restrictions have no impact with respect to compression. The *size* of G is defined as $|G| := \sum_{(A,g) \in P} |g|$, and similarly the edge and node sizes: $|G|_E := \sum_{(A,g) \in P} |g|_E$ and $|G|_V := \sum_{(A,g) \in P} |g|_V$. We often write $p : A \to g$ for a rule $p = (A, g)$ and call A the left-hand side and g the right-hand side $\mathsf{rhs}(p)$ of p. We call symbols in Σ *terminals*. Consequently an edge is called *terminal* if it is labeled by a terminal and *nonterminal* otherwise. To derive a nonterminal edge e using a rule $A \to h$ in a graph g we remove e from g, add a disjoint copy h to g and merge the ith external node of h with the ith node of $\mathsf{att}_g(e)$. For an example of a derivation, see Figure 1.

An HR grammar G is called *straight-line* (SL-HR grammar) if (1) the relation $\leq_{\mathsf{NT}} = \{(A_1, A_2) \mid \exists g : (A_1, g) \in P, \exists e \in E_g : \mathsf{lab}(e) = A_2\}$ is acyclic and (2) for every nonterminal $A \in N$ there exists exactly one rule $(A, g) \in P$. Note that SL-HR grammars always derive exactly one hypergraph (up to isomorphism). As the right-hand side for a nonterminal is unique in SL-HR grammars, we denote the right-hand side of $p = (A, g)$ by $\mathsf{rhs}(A)$. The height of an SL-HR grammar $\mathsf{height}(G)$ is the height of $\leq_{\mathsf{NT}}$. We now present a method to assign precise node IDs by imposing an order on the nonterminal edges. We use numbers from 1 to $m = |V_S|$ for the node IDs in the start graph. Now let the nonterminal edges be ordered. Then, when applying the rules in this order, we assign the next available IDs $m + 1$, $m + 2$,... to the nodes created in the hypergraph, in the same order as they are given in the right-hand side of the rule. Doing so yields a unique hypergraph out of the many isomorphic options in $L(G)$. We denote this hypergraph by $\mathsf{val}(G)$. We denote the hypergraph derived by a single edge e in this way by $\mathsf{val}(e)$.

In the following we often say *grammar* instead of *SL-HR grammar* and *graph* instead of *hypergraph*.

III. GraphRePair

We present a generalization to graphs of the RePair compression scheme. Let us first explain the classical RePair compressor for strings and trees. Consider as an example the string *abcabcabc*: it contains occurrences of the digrams *ab* (3 times), *bc* (3 times), and *ca* (2 times). If *ab* is replaced by A then we obtain *AcAcAc*. In the next step *Ac* is replaced by B, to obtain this grammar $\{S \to BBB, B \to Ac, A \to ab\}$. To compute in linear time such a grammar from a given string requires a set of carefully designed data structures. The input string is represented as a doubly linked list. Additionally a list of active digrams (digrams that occur at least twice) is maintained. Every entry in the list of active digrams points to an entry in a priority queue PQ of size $\sqrt{n}$ (with n being the size of the input string) containing doubly linked lists. The list with priority i in PQ contains every digram that occurs i times, the last list contains every digram occurring $\sqrt{n}$ or more times. The list items also contain links to the first occurrence of the respective digram. This queue is used to find the most frequent digram in constant time. Larsson and Moffat [15] proved that $\sqrt{n}$-length guarantees constant time. All these data structures are updated whenever an occurrence is removed. Consider removing one occurrence of *ab* in the example above: when doing so, one occurrence of *bc* and possibly *ac* needs to be removed from the list. On the other hand, the new occurrences of *Ac* and possibly *cA* are created.

An even smaller grammar than the one above can be obtained through *pruning*, which removes nonterminals that are referenced only once, i.e., the B-rule becomes $B \to abc$. On trees, a digram consists of a node and one of its children. Such a digram has, in a binary tree, at most three "dangling edges". Dangling edges in context-free tree grammars are represented by *parameters* of the form $y_1, \ldots, y_k$. The number k is the *rank* of the rule (digram). E.g. the A-rule in Figure 3 represents a digram of rank 1, while the B-rule represents a digram of rank 3. Keeping the rank small is desirable as it impacts further algorithms on the grammar [26]. Therefore, TreeRePair has a user-defined "maxRank" parameter.

Definition 2. A *digram* over Σ is a hypergraph $d \in \mathsf{HGR}(\Sigma)$, with $E_d = \{e_1, e_2\}$ such that (1) for all $v \in V_d$, $v \in \mathsf{att}_d(e_1)$ or $v \in \mathsf{att}_d(e_2)$, and (2) there exists a $v \in V_d$ such that $v \in \mathsf{att}_d(e_1)$ and $v \in \mathsf{att}_d(e_2)$.

Every possible digram over undirected, unlabeled edges is shown in Figure 2. As a further example, the right-hand sides of the two A rules in Figure 4 are digrams. Note that both grammars in the figure generate the graph on the left. However, they differ in size: the grammar in the middle has size 12, while the grammar on the right has size 9 (recall that simple edges have size 1, even if they are nonterminal).

A. The Algorithm

As mentioned before, RePair replaces a digram that has the largest number of non-overlapping occurrences.

Definition 3. Let $g, d \in \mathsf{HGR}(\Sigma)$ such that d is a digram. Let e_1^d, e_2^d be the two edges of d. Let $o = \{e_1, e_2\} \subseteq E_g$ and let

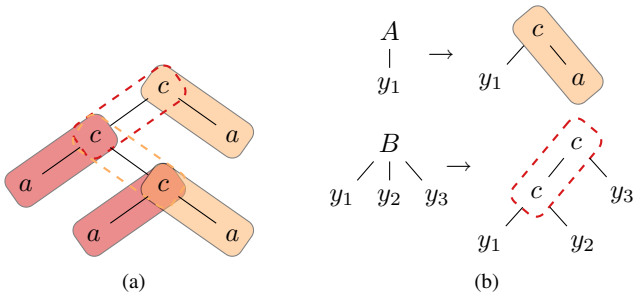

Fig. 3. Different digrams TreeRePair considers in a tree and their replacement rules.

V_o be the set of nodes incident with edges in o. Then o is an *occurrence* of d in g, if there is a bijection $\psi : V_o \to V_d$ such that for $i \in \{1, 2\}$ (1) $\psi(v) \in \mathsf{att}_d(e_i^d)$ if and only if $v \in \mathsf{att}_g(e_i)$, (2) $\mathsf{lab}_d(e_i^d) = \mathsf{lab}_g(e_i)$ and (3) $\psi(v) \in \mathsf{ext}_d$ if and only if there exists an edge $e \in E_g$ with $e \neq e_1$, $e \neq e_2$ and $v \in \mathsf{att}_g(e)$.

The first two conditions of this definition ensure that the two edges of an occurrence induce a graph isomorphic to d. The third condition requires that every external node of d is mapped to a node in g that is incident with other edges. Thus, the edges marked in the graph on the left of Figure 4 are an occurrence of the digram in the right grammar, but not an occurrence of the digram in the middle grammar. We call the nodes in V_o that are mapped to external nodes of d, *attachment nodes* of o, and the ones mapped to internal nodes, *removal nodes* of o. Two occurrences o_1, o_2 of the same digram d are called *overlapping* if $o_1 \cap o_2 \neq \emptyset$. Otherwise they are *non-overlapping*. If there are at least two non-overlapping occurrences of d in a graph g, we call d an *active digram*.

Let X be a symbol of rank k and d a digram of rank k. The *replacement of an occurrence o of d in g by X* is the graph obtained from g by removing the edges in o from g, removing the removal nodes of o, and adding an edge labeled X that is attached to the attachment nodes of o, in such a way that applying the rule $X \to d$ yields the original graph. Consider again Figure 4: the start graph of the right grammar is the replacement of the shaded occurrence of d in the left graph by A (where d is $\mathsf{rhs}(A)$ of the grammar on the right).

Given a graph g gRePair performs these steps:

1) Let $G = (N, P, S)$ be a grammar with $N = P = \emptyset$ and $S = g$.
2) Determine a list of non-overlapping occurrences for every digram appearing in g.
3) Select a most frequent digram d.
4) Let A be a fresh nonterminal of rank $\mathsf{rank}(d)$. Replace every occurrence o of d in S by an A-edge.
5) Let $N = N \cup \{A\}$ and $P = P \cup \{A \to d\}$.
6) Update the occurrence lists.
7) If there are active digrams in S: repeat from Step 3.
8) Prune the grammar.

As an additional step after Step 3 finishes, we connect the disconnected components of the graph by virtual edges and run the algorithm again before pruning (and then remove the virtual edges from the grammar). This improves the compression on graphs with disconnected components. We provide more details on Steps 2, 6, and 8.

1) Counting occurrences: (Step 2) We aim to find a set of non-overlapping occurrences for every digram that occurs in g, that is of maximal size. As stated in the Introduction, the most efficient way of doing this we are aware of needs $O(|V|^2|E|)$ time. Thus we approximate. Let ω be an order on the nodes of g. We traverse the nodes of g in this order, and at every node iterate through occurrences centered around this node. The node order ω heavily influences the compression behavior. Consider the graph in Figure 5. We want to find the non-overlapping occurrences of the digram in Figure 5d. Note that all three nodes are external, but their order is not important in this example. Figure 5a shows the non-overlapping occurrences found if we start in the central node of the graph. Using the DFS-type order starting at a different node given by the numbers in Figure 5b, three occurrences are determined. Using the "jumping" order in Figure 5c, a maximum set of four non-overlapping occurrences is found. Note that for strings and trees, maximum sets of non-overlapping occurrences can be obtained by simple orders (namely, left-to-right and post order), and straightforwardly assigning occurrences in a greedy way. Implementation details of this step are explained in Section III-C1.

2) Updating occurrence lists: (Step 6) Let o be an occurrence of d that is being replaced and let F be the set of edges in g that are incident with the attachment nodes of o. Removing o from the graph can only affect the occurrence lists of digrams that have occurrences using edges in F. In particular, for the two edges in o (e_1 and e_2) reduce the count of every digram by one for which $\{e_i, e\}$ ($i \in \{1, 2\}, e \in F$) appears in an existing occurrence list. After the replacement let e' be the new A-labeled nonterminal edge in g. Then every pair of edges $\{e', e\}, e \in F$ is an occurrence of a digram, and is thus inserted into the appropriate occurrence list. The last step has again complexity issues. Let k be the sum of degrees of all attachment nodes of o. Then there are $O(k)$ pairs of edges to be considered as occurrences with the new nonterminal edge. This is not a problem in itself, but consider the following situation: let there be an attachment node v with degree k. Further, let every one of the k edges around v be part of a distinct occurrence of the digram d being replaced. As explained, when replacing one of these occurrences, the other $k - 1$ edges are considered as occurrences with the new nonterminal. Now however, when replacing the next one, the remaining $k - 2$ edges have to be considered again. Thus, during all the replacement steps, we would again need to consider $O(k^2)$ occurrences. Therefore this update needs to be done in constant time. See Section III-C1 for details how we solved this issue.

3) Pruning: (Step 8) This phase removes every rule from the grammar that does not contribute to the compression. For a nonterminal A of rank n we define $\mathsf{handle}(A) = (\{v_1, \ldots, v_n\}, \{e\}, \mathsf{lab}(e) = A, \mathsf{att}(e) = v_1 \cdots v_n, \mathsf{ext} = v_1 \cdots v_n)$, as a minimal graph with a nonterminal A-edge. The size of $\mathsf{handle}(A)$ is precisely the size a nonterminal edge adds to a graph. We then define the *contribution of A* as

$$\mathsf{con}(A) = |\mathsf{ref}(A)| \cdot (|\mathsf{rhs}(A)| - |\mathsf{handle}(A)|) - |\mathsf{rhs}(A)|,$$

where $\mathsf{ref}(A) = |\{e \in E_S \mid \mathsf{lab}(e) = A\}| + \sum_{B \in N} |\{e \in E_{\mathsf{rhs}(B)} \mid \mathsf{lab}(e) = A\}|$ is the number of edges labeled A in the

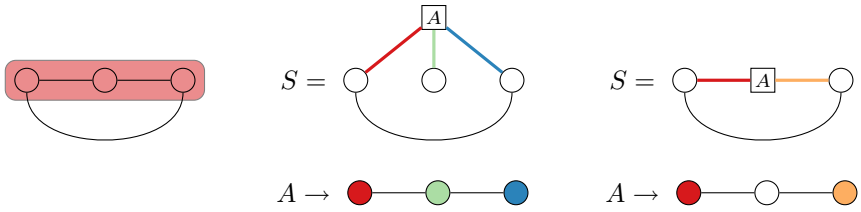

Fig. 4. Two ways of replacing a pair of edges by a nonterminal. Only the one on the right is considered by gRePair in this case.

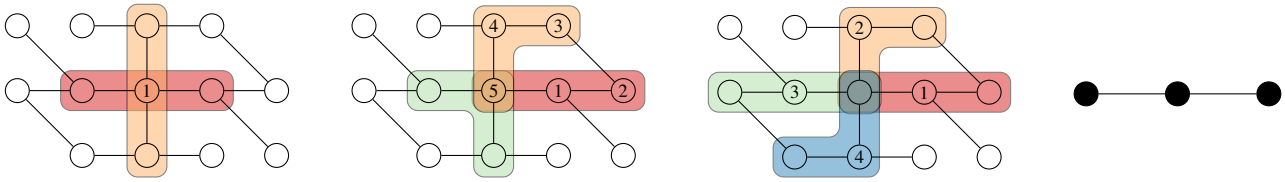

Fig. 5. Three different traversals visiting the nodes in the numbered order to find occurrences of the digram given in (d). The occurrences found for each traversal are marked by differently colored boxes.

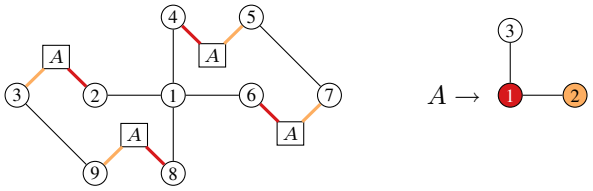

Fig. 6. Example of a hyperedge replacement grammar.

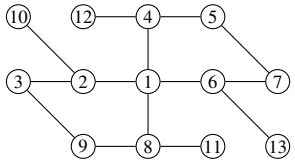

Fig. 7. Unique result of the derivation of the grammar in Figure 6 when ordering the nonterminal edges from left to right.

grammar. The contribution of A counts by how much the size of the grammar changes when every instance of the nonterminal is derived, i.e., it measures how much A contributes towards compression. If $\mathsf{con}(A) > 0$ then we say that A *contributes towards compression*. The grammar in Figure 6represents the graph of Figures 5a–5c. Here, the A-rule has $\mathsf{con}(A) = 4 \cdot (5 - 3) - 5 = 3$ and thus contributes to the compression. The reader may verify that the sizes of this grammar and the graph (given in Figure 7, with the IDs assigned as explained at the end of Section II ordering the nonterminals from "left" to "right") differ by exactly three. Note that we cannot just remove every rule with a contribution of 0 or less: as we remove rules, the contribution of other nonterminals might change as edges are added or deleted.

The effectiveness of pruning depends on the order in which the nonterminals are considered. Finding an optimal order is a complex optimization problem as mentioned in [16, Section 3.2]. For TreeRePair, a bottom-up hierarchical order works well in practice. We use a similar approach. First every nonterminal A with $\mathsf{ref}(A) = 1$ is removed, because, by definition, they do not contribute towards compression. To remove A we apply its rule to each A-edge in the grammar and remove the A-rule. Then we traverse the nonterminals in bottom-up $\leq_{\mathsf{NT}}$-order (see Preliminaries), removing each nonterminal with $\mathsf{con}(A) \leq 0$.

B. Important Parameters

In this section we describe some parameters of our algorithm that influence the compression ratio. Their effect is evaluated experimentally in Section IV.

1) Node order: The node order strongly influences the digram counting. As we cannot guarantee to find a maximal set of non-overlapping occurrences for every digram, the node order becomes the main factor in the quality of this set. We evaluate these orders: *natural order* uses the node IDs as given, *BFS* order follows a breadth-first traversal, and *FP* computes a fixpoint on the node neighborhoods starting from the degrees (as a fourth order, we consider *FP0*, which is a degree order):

For a graph g let $c_i : V_g \to \mathbb{N}$ be a family of functions that color every node with an integer. We first define $c_0(v) = d(v)$, where $d(v)$ is the degree of v. Now map every node v to the tuple $f_0(v) = (c_0(v), c_0(v_1), \ldots, c_0(v_n))$, where $v_1, \ldots, v_n$ are the neighbors of v ordered by their values in c_0. Sort these tuples lexicographically and let $c_1(v)$ be the position of $f_0(v)$ in this lexicographical order. This process is iterated until $c_{i+1} = c_i$. Now c_i implies an order of the nodes by defining $v < u$ iff $c_i(v) < c_i(u)$. This computation of the order works for undirected, unlabeled graphs, but can be straightforwardly extended to directed labeled graphs. We call this order FP. Figure 8 shows an example of the FP-order. The graph on the left is annotated by c_0, the graph in the middle shows f_0, which is then ordered lexicographically to get c_1 on the right. This is the fixpoint for this graph. Note that it is not necessarily total and thus also implies an equivalence relation on the nodes ($v \cong_{\mathrm{FP}} u$ iff $c_i(v) = c_i(u)$). The number of equivalence classes of $\cong_{\mathrm{FP}}$ has an interesting correlation with the compression ratio, as discussed in Section IV-B2.

2) Maximum rank: This specifies the maximal rank of a digram (and thus the maximal rank of a nonterminal edge) the compressor considers. Digrams with a higher rank are ignored and not counted. It was shown already for TreeRePair [16] that

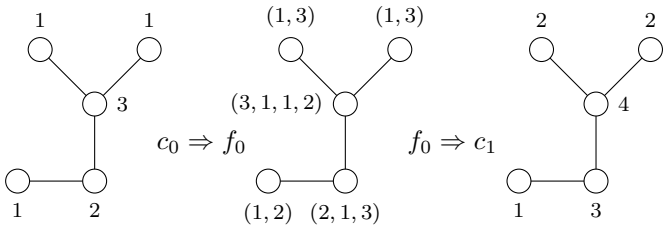

Fig. 8. Computing the FP-order of a small graph.

choosing this parameter too high or too small can have strong effects on compression (in both directions).

C. Implementation Details

In this section we describe some of the technical details of our implementation. We outline occurrence counting and the involved data structures, and describe our output format.

1) Compression: We focus on the first phase in this section, because the implementation of the pruning phase is straightforward. There are two details we want to discuss. The first one is the counting of occurrences centered around a node v. As mentioned in the previous section, there are $O(k^2)$ possible edge pairs to consider, if k is the degree of v, but we want to only consider $O(k)$ many. The second one is the data structure used to maintain occurrences and allowing for quick updates.

Occurrence lists: Consider first the case of a graph without edge labels and directions. Let node v have degree k, i.e., there are $O(k^2)$ edge pairs that are occurrences of some digram. But, after inserting one of them into the occurrence list, we effectively take the two edges involved out of further consideration, because every other occurrence using one of them would overlap with this one. Let E be the edges incident with v that have not been counted as occurrences yet. We partition E into two sets $E_1 = \{e_1, \ldots, e_n\}$ and $E_2 = \{f_1, \ldots, f_m\}$ where $m - n \in \{0, 1\}$. We then add $\mathsf{Occ}(E_1, E_2) = \{\{e_i, f_i\} \mid 1 \leq i \leq n\}$ as the $O(k)$ occurrences around v to the list. Note that only if all occurrences around v are occurrences of the *same* digram, this procedure guarantees to produce a maximum non-overlapping set of occurrences around v.

From here, adding labels (or directions, which can be viewed as labels) is straightforward. For two labels σ_1 and σ_2 let $E_{\sigma_1,\sigma_2}(v)$ be the set of edges incident with v labeled σ_1 and not yet counted in an occurrence with an edge labeled σ_2. Then for distinct symbols σ_1 and σ_2 count the occurrences $\mathsf{Occ}(E_{\sigma_1,\sigma_2}(v), E_{\sigma_2,\sigma_1}(v))$ and for every σ count the occurrences where both edges have the same symbol by splitting $E_{\sigma,\sigma}(v)$ as above. This takes $O(|\Sigma|^2)$ time, but we expect $|\Sigma|$ to be comparatively small, so this is not an issue.

A similar situation arises when updating the occurrence list after removing an occurrence. As mentioned in Section III-A2, after inserting the new nonterminal edge e' we need to select an edge e from the set of neighboring edges F in constant time. Our implementation does this by storing a list of available edges for every combination of edge labels attached to every node of the graph. For every edge label the first edge e in the respective list is selected to create the occurrence $\{e', e\}$. This takes $O(|\Sigma|)$ time.

Data Structures: Our data structures are a direct generalization to graphs of the data structures used for strings [15] and trees [16, Figure 11]. The occurrences are managed using doubly linked lists for every active digram. Of importance is a priority queue, which uses the frequency of a digram as the priority. Following Larsson and Moffat [15] the length of this queue is chosen as $\sqrt{n}$, where $n = |E|$.

2) Grammar Representation: We encode the start graph and the productions in different ways. As an example, consider again the grammar in Figure 6. The start graph is encoded using k^2-trees [21], using $k = 2$ as this provides the best compression. This data structure partitions the adjacency matrix into k^2 squares and represents it in a k^2-ary tree. Consider the left adjacency matrix in Figure 9. The 9×9-matrix is first expanded with 0-values to the next power of two; i.e., 16×16. If one partition has only 0-entries, a leaf labeled 0 is added to the tree. This happens for the 3rd and 4th partition in this case (the partitions are numbered left to right, top to bottom). Thus the 3rd and 4th child of the root are 0-leafs. The other two have at least one 1-entry, therefore inner nodes labeled 1 are added and the square is again partitioned into k^2 squares. This is continued at most until every square covers exactly one value. At this point the values are added to the tree as leafs. As we need to consider edges with different labels, we also use a method similar to the representation of RDF graphs proposed in [8]. Let $E_\sigma \subseteq E$ be the set of all edges labeled σ. For every label σ appearing in S we encode the subgraph (V, E_σ). If $\mathsf{rank}(\sigma) = 2$, then this is encoded as an adjacency matrix, otherwise we use an incidence matrix. The resulting matrices are encoded as k^2-trees. Figure 9 is an example with two edge labels. Note that this example only uses edges of rank 2. For a hyperedge e, the incidence matrix only provides information on the set of nodes attached to e, but not the specific order of $\mathsf{att}(e)$. For this reason we also store a permutation for every edge to recover $\mathsf{att}(e)$. We count the number of distinct such permutations appearing in the grammar and assign a number to each. Then we store the list encoded in a $\lceil \log n \rceil$-fixed length encoding, where n is the number of distinct permutations.

For the productions we use a different format, as we expect the right-hand-sides to be very small graphs. We store an edge list for every production, encoding the nodes using a variable-length δ-code [27]. One more bit per node is used to mark external nodes. As the order of the external nodes is also important, we make sure that the order induced by the IDs of the external nodes is the same as the order of the external nodes. Furthermore, due to the pruning step, productions can have more than two edges, so every production begins with the edge count (again, using δ-codes). For every edge we first use one bit to mark terminal/nonterminal edges, then store the number of attached nodes, followed by the δ-codes of the list of IDs. Finally, we also use a δ-code for the edge label. For the production in Figure 6 this leads to the following encoding:

$\delta(2)$	two edges
$0\delta(2)$	edge is terminal (0), has two nodes
$1\delta(1)1\delta(2)\delta(1)$	nodes 1 (external) and 2 (external), label 1
$0\delta(2)$	next edge is terminal, has two nodes
$1\delta(1)0\delta(3)\delta(1)$	nodes 1 (external) and 3 (internal), label 1

This is a bit sequence of length 28.

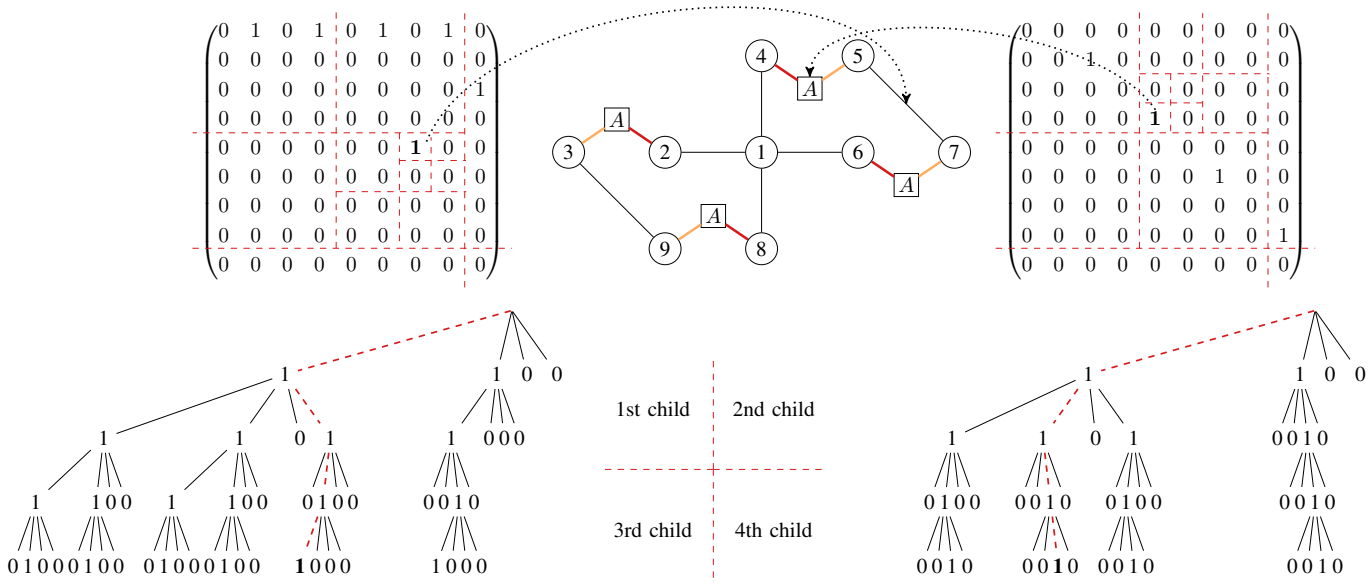

Fig. 9. Start graph (middle): terminal edges (left) and nonterminal edges (right) and their k^2-tree representations (below) with $k = 2$.

Note that the grammar only produces an isomorphic copy of the original graph. We can, however, produce a mapping from the new node IDs to the original ones, as it always produces the same isomorphic copy. This can be used to create a mapping $\psi' : V_{\mathsf{val}(G)} \to D$, such that the grammar represents the same data as the original graph.

IV. Experimental results

We implemented a prototype in Scala (version 2.11.7) using the Graph for Scala library[1] (version 1.9.4). The experiments are conducted on a machine running Scientific Linux 6.6 (kernel version 2.6.32), with 2 Intel Xeon E5-2690 v2 processors at 3.00 GHz and 378 GB memory. As we are only evaluating a prototype, we do not mention runtime or peak memory performance, as these can be improved by a more careful implementation. We compare to the following compressors:

- k^2-tree, for which we use our own Scala-implementation following the description in [21], using the same binary format.
- The list merge (LM) algorithm by Grabowski and Bieniecki [20]. We use 64 for their chunk size parameter, as in their paper.
- The combination of dense substructure removal [23] with k^2-tree by Hernández and Navarro [22] (HN). For the parameters to the algorithm we use $T = 10$, $P = 2$, and $ES = 10$, which are the paremeters their experiments show to provide the best compression.

The latter two implementations were provided by their authors. We also experimented with RePair on adjacency lists by Claude and Navarro [19], but omit the results here, because the compression performance generally was superseded by another compared method.

TABLE I. Network graphs

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert[\cong_{\mathrm{FP}}]\rvert$
CA-AstroPh	18,772	396,160	14,742
CA-CondMat	23,133	186,936	17,135
CA-GrQc	5,242	28,980	3,394
Email-Enron	36,692	367,662	5,805
Email-EuAll	265,214	420,045	28,895
NotreDame	325,729	1,497,134	118,264
Wiki-Talk	2,394,385	5,021,410	566,846
Wiki-Vote	7,115	103,689	5,806

As common in graph compression, we present the compression ratios in *bpe* (bits per edge). Note that our results are not perfectly comparable, as we do not include the space required to retain the original node IDs. However, in particular for RDF we do not consider this a limitation, as explained in Section IV-C2.

A. Datasets

We use three different types of graphs: network graphs (Table I), RDF graphs (Table II), and version graphs (Table III). Every table lists the numbers of nodes and edges for each graph and the number of equivalence classes of $\cong_{\mathrm{FP}}$ (see Section III-B1). For RDF graphs we also list how many distinct edge labels (i.e., predicates) the data set uses. Two of the version graphs also have labeled edges. We give a short description of every graph used: the network graphs are from the Stanford Large Network Dataset Collection[2] and are unlabeled. They are communication networks (Email-EuAll, Wiki-Vote, Wiki-Talk), a web graph (NotreDame) and Co-Authorship networks

[1] http://www.scala-graph.org/

[2] http://snap.stanford.edu/data/index.html

TABLE II. RDF GRAPHS

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Sigma\rvert$	$\lvert[\cong_{FP}]\rvert$
1 Specific properties en	609,014	819,764	71	236,235
2 Types ru	642,340	642,364	1	79
3 Types es	818,657	819,780	1	336
4 Types de with en	618,708	1,810,909	1	335
5 Identica	16,355	29,683	12	14,588
6 Jamendo	438,975	1,047,898	25	396,725

TABLE III. VERSION GRAPHS

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Sigma\rvert$	$\lvert[\cong_{FP}]\rvert$
Tic-Tac-Toe	5,634	10,016	3	9
Chess	76,272	113,039	12	74,592
DBLP60-70	24,246	23,677	1	2,739
DBLP60-90	658,197	954,521	1	207,305

(CA-AstroPh, CA-CondMat, CA-GrQc). Even if they were advertised as undirected, we considered all of them to be lists of directed edges, to improve the comparability with other methods.

The RDF graphs we use mostly come from the DBPedia project[3], which is an effort of representing ontology information from Wikipedia. We use *specific mapping-based properties* (English), which contains infobox data from the English Wikipedia and *mapping-based types*, which contains the rdf:types for the instances extracted from the infobox data. We use three different versions of the latter: types for instances extracted from the Spanish and Russian Wikipedia pages that do not have an equivalent English page, and types for instances extracted from the German Wikipedia pages that do have an equivalent English page. The Identica-dataset[4] represents messages from the public stream of the microblogging site identi.ca. Its triples map a notice or user with predicates such as creator (pointing to a user), date, content, or name. The Jamendo-dataset[5] is a linked-data representation of the Jamendo-repository for Creative Commons licensed music. Subjects are artists, records, tags, tracks, signals, or albums. The triples connect them with metadata such as names, birthdate, biography, or date.

Version graphs are disjoint unions of multiple versions of the same graph. Here, Tic-Tac-Toe represents winning positions, Chess the legal moves in the respective games[6]. The files contain node labels from a finite alphabet, which we ignore here. DBLP60-70 and DBLP60-90 are co-authorship networks from DBLP, created from the XML[7] file by using author IDs as nodes and creating an edge between two authors who appear as co-authors of some entry in the file. To make version graphs, we created graphs containing the disjoint union of yearly snapshots of the co-authorship network.

B. Influence of Parameters

We evaluate how the different parameters for our compressor affect compression. For these experiments every parameter except the one evaluated is fixed for the runs. Note that this sometimes leads to situations where none of the results in a particular experiment represents the best compression our compressor is able to achieve for the given graph. The parameters evaluated are the maximum rank of a nonterminal and the node order.

[3]http://wiki.dbpedia.org/Downloads2015-04

[4]http://www.it.uc3m.es/berto/RDSZ/

[5]http://dbtune.org/jamendo/

[6]Both from http://learnlab.uta.edu/old_site/subdue/index.html

[7]http://dblp.uni-trier.de/xml/ (release from August 1st, 2015)

TABLE IV. RESULTS FOR DIFFERENT VALUES OF MAXRANK (COMPRESSION IN BPE).

	2	3	4	5	6	7	8
Email-EuAll	6.66	6.69	**6.42**	7.07	7.33	7.55	7.36
NotreDame	**4.84**	4.90	5.19	5.14	6.13	7.10	6.69
CA-AstroPh	16.94	**16.75**	16.77	16.75	17.44	19.42	18.36
CA-CondMat	18.82	17.73	**17.40**	18.47	18.84	20.26	19.83
CA-GrQc	13.65	13.31	**13.20**	14.30	14.91	15.04	14.93
Email-Enron	**10.21**	10.74	10.28	10.79	11.62	13.29	11.53

1) Maximum Rank: We tested maxRank values from 2 up to 8, the results of a subset of six graphs are given in Table IV, as compression in bpe. We did some tests for higher values (up to 16) but only got worse results. We did not evaluate values higher than 16. The best results are marked in bold. In most cases the best result was either achieved with a setting of 2 or with a value of 4. Even in the cases where a maximal rank of 4 does not yield the best result, the difference is less than 1 bpe. We therefore conclude that a value of 4 is a good compromise for our data set.

2) Node Order: Recall from Section III-B1 that the FP-order is a fixed point computation starting from the node degrees. As this is an iterative process, it can be terminated at any point. We were interested how much difference finding a fixpoint makes compared to using just the node degrees (which we call FP0). Figure 10 shows the compression ratio of a selection of graphs under the different node orders. The selection aims to be representative for the graphs of the types we evaluated: CA-graphs behave similar to CA-AstroPh, version graphs similar to DBLP60-70, and the RDF graphs similar to Specific properties en. The other graphs in the figure are chosen because they are outliers in their respective category. Our FP-order achieves the best result on most of the graphs, but the different orders tend to only have surprisingly little impact. On RDF graphs the order generally had only marginal impact: the best and worst results usually are within 0.5 bpe of each other. The Jamendo graph presents an exception here, with the natural order being about 1 bpe better than the closest other result. Version graphs however benefit hugely from the FP-order. This shows that two or more versions of the same graph are similarly ordered in the FP-order, increasing the likelihood of the compressor of recognizing repeating structures.

There is another interesting observation about the FP-order, or in particular the equivalence relation $\cong_{FP}$. It is likely that isomorphic nodes are equivalent in this relation. This implies that graphs with a low number of equivalence classes should compress well, as they would have many repeating substructures. Figure 11 shows this correlation. There is no graph in the lower right corner, i.e., there is no graph with a low number of equivalence classes and bad compression.

C. Comparison with other Compressors

We compare gRePair with several other compressors. Note that we compare RDF graph compression only against the k^2-tree-method, as LM and HN have not been extended to

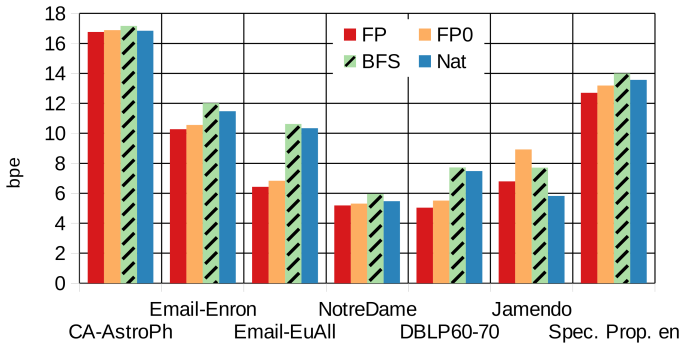

Fig. 10. Performance of gRePair under different node orders.

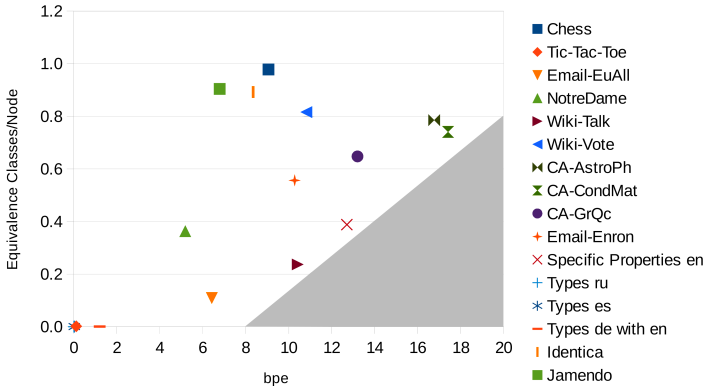

Fig. 11. Correlation between equivalence classes of $\cong_{FP}$ and compression.

RDF graphs. While these algorithms all work as in-memory data structures, they produce outputs with file sizes comparable to the in-memory representation. We therefore measure the compression performance based on the file size. We also want to give an idea of the compression ratio for the graphs according to our size definition. On average we achieve a compression ratio ($\frac{|G|}{|g|}$) of 68% for network graphs, 35% for RDF, and 24% for version graphs. The parameters we choose for gRePair are maxRank $= 4$ and the FP-order, both being generally the best, or close to the best, choice for our dataset. We note first that in most results the majority of the file size of gRePairs output ($> 90\%$) is for the k^2-tree representation of the start graph.

1) Network Graphs: Our results on network graphs compared to k^2-tree, LM, and HN are summarized in Figure 12. We improve on the plain k^2-tree-representation on all graphs but NotreDame. However, our results are generally worse than LM and HN, with Email-EuAll and CA-GrQc being the only exceptions. That being said, the HN-method can be combined with our compressor, using their dense substructure removal as a preprocessing step. This combination then achieves the smallest bpe-values for the three CA-graphs.

2) RDF Graphs: The RDF format lists triples (s, p, o) of subject, predicate, and object. These can be URIs or other values, but in any case they tend to be big. A common practice (see e.g. [8], [28], [29]) is to map the possible values to integers using a dictionary and represent the graph using triples of integers. A triple (s, p, o) then represents an edge from s to o labeled p. As in this way dictionary and graph are separate entities, we only focus on compressing the graph. Any method for dictionary compression can be used to additionally compress

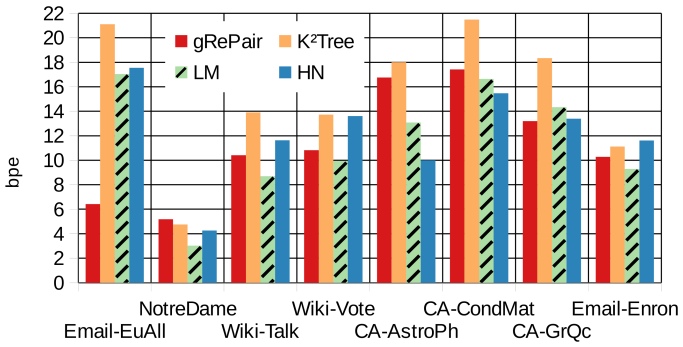

Fig. 12. Network graph comparison of gRePair with three other compressors.

TABLE V. RESULTS ON RDF GRAPHS (SIZE IN KB)

	1	2	3	4	5	6
gRePair	1,271	1	3	267	30	872
k^2-tree	2,731	590	938	1,119	52	988

the dictionary (e.g. [29]). In our comparison with k^2-trees we therefore omit the space necessary for the dictionary in both cases. Also note that we can easily reorder the dictionary, making RDF graphs a case where recovering only an isomorphic copy is no limitation.

The way of extending the k^2-tree-method to compress RDF graphs is similar to our way of encoding the start graph: one adjacency matrix is created for every edge label and then encoded as a separate k^2-tree. This was shown to be effective in [8], where they compare to four other RDF graph representations and achieve smaller representations than these.

Our results in comparison to k^2-tree are given in Table V. We greatly improve on the representation. Occasionally (for the instance types graphs in particular) we are able to produce a representation that is orders of magnitude smaller than the k^2-tree-representation. For these two graphs in particular, this is due to the majority of their nodes being laid out in a star pattern: few hub nodes of very high degree are connected to nodes, most of which are only connected to the hub node. Structures like these are compressed well by gRePair.

3) Version Graphs: We describe several experiments over version graphs. First we study how the compressor behaves given a high number of identical copies of the same simple graph. The graph in this case is a directed circle with four nodes and one of the two possible diagonal edges. Figure 13 shows the results of this experiment for identical copies starting from 8 going in powers of 2 up to 4096. Clearly, gRePair is able to compress much better in this case ("exponential compression"), while the file size of other methods rises with roughly the same gradient as the size of the graph. Note that both axes in this graph use a logarithmic scale: in this case, gRePair produces a representation that is orders of magnitude smaller.

Except for identical copies of rather simple graphs, however, we cannot expect to achieve exponential compression on version graphs. This only works, if gRePair consistently compresses the same (i.e., isomorphic) substructures in the same way. Our FP-ordering approximates a test for isomorphism, but of course cannot achieve this for large graphs. Figure 14 shows a comparison on the compression of a version graph from the

TABLE VI. RESULTS ON VERSION GRAPHS (COMPRESSION IN BPE)

	TTT	Chess	DBLP60-70	DBLP60-90
gRePair	0.12	9.06	9.54	13.39
k^2-tree	9.62	13.10	15.78	20.80
LM	-	-	16.44	19.32
HN	-	-	16.65	18.26

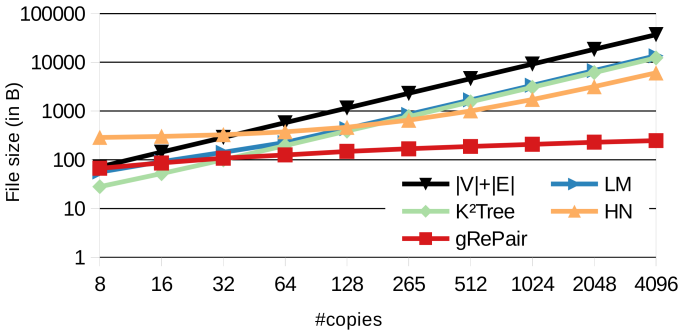

Fig. 13. Compression of disjoint unions of the same synthetic graph with 4 nodes and 5 edges.

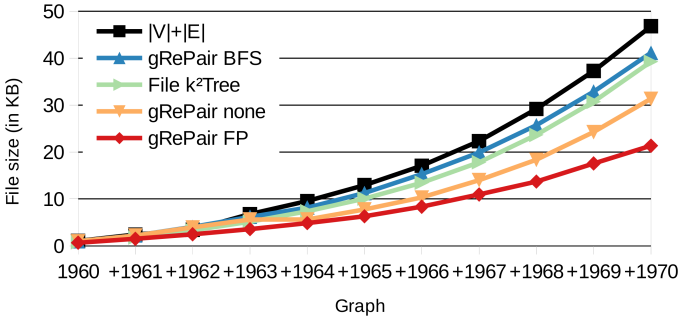

Fig. 14. Using different node orders for the compression of a version graph (yearly snapshots of the DBLP co-authorship network from 1960 to 1970).

DBLP co-authorship network. We started with a co-authorship network including publications from 1960 and older. To this graph we then add versions with the publications from 1961, 1962,... until 1970 and compress the graphs obtained in this way. The comparison shows that, using the FP-ordering, our method achieves better compression than using other orders. Note that the results for BFS or random ordering are much closer to k^2-trees. Our full results for version graphs are given in Table VI. Note that we compare Tic-Tac-Toe and Chess only against k^2-tree, because these graphs have edge labels. The results show that gRePair compresses version graphs well. Recall, that TTT and Chess are not compared against LM/HN, because these graphs have edge labels.

V. QUERY EVALUATION

In this section we discuss two types of queries that can be performed over SL-HR grammars: neighborhood queries and speed-up queries. *Neighborhood queries* allow to traverse the edges of a graph (in any direction). Using them, any arbitrary graph algorithm can be performed on the compressed representation given by an SL-HR grammar. However, this comes at a price: a considerable slow-down is to be expected in comparison to running over an uncompressed graph representation. In contrast, *speed-up queries*, as their name suggests, can run faster on an SL-HR grammar than on an uncompressed graph representation. Examples of speed-up queries are counting the number of connected components of the graph, checking regular path properties in the graph, or checking reachability between two nodes. These queries can be evaluated in one pass through the grammar, and hence allow speed-ups proportional to the compression ratio. Strictly speaking, these queries take time quadratic in the size of the grammar (see Proposition 5). Therefore we present a true linear time (in $|G|$) algorithm for *reachability queries*.

The results in this section have not been implemented. Over grammar-compressed trees, the performance of simple speed-up queries is evaluated in [30].

Neighborhood Queries: First, we define some necessary terms for neighborhood: For a node $v \in V_g$ of a hypergraph g we denote by $N(v) = \{u \in V_g \mid \exists e \in E_g : v \in \mathsf{att}_g(e)$ and $u \in \mathsf{att}_g(e)\}$ the *neighborhood of* v. For simple graphs we also define $N^+(v) = \{u \in V_g \mid \exists e \in E_g : \mathsf{att}(e) = uv\}$ and $N^-(v) = \{u \in V_g \mid \exists e \in E_g : \mathsf{att}(e) = vu\}$, the *incoming* and *outgoing neighborhoods of* v, respectively. Furthermore we let $E(v) = \{e \in E_g \mid v \in \mathsf{att}(e)\}$ be the set of edges incident with v.

Let $G = (N, P, S)$ be an SL-HR grammar. We assume that every right-hand side in P contains at most two nonterminal edges. Recall that the nodes in the start graph S are numbered $1, .., m$, and that there is an order on the nonterminal edges $e_1, \ldots, e_\ell$ in S so that the nodes in $g_1 = \mathsf{val}(e_1)$ are numbered $m+1, \ldots, m+v_1$, where $v_1 = |g_1|_V)$, and similarly, nodes in $g_i = \mathsf{val}(e_i)$ are numbered from $k = m + \sum_{j=1}^{i-1} v_j$ to $k + v_i$. Given a node ID, i.e., a number in $k \in \{1, \ldots, |\mathsf{val}(G)|_V\}$, computing its incoming neighbors consists of two steps: (1) compute a *grammar representation* (*G-representation*) of k, i.e, a path in the derivation tree of G that "derives the node k". Such a path is of the form wv where w is a, possibly empty, string of the form $e_0 e_1 \cdots e_n$. If w is empty, then v must be a node in S. If not, then e_0 is a nonterminal edge of S. If A_1 is the label of e_0, then e_1 is a nonterminal edge in $\mathsf{rhs}(A_1)$ labeled A_2, etc. Finally, v is an internal node in $\mathsf{rhs}(A_n)$. Let ℓ be the number of nonterminal edges in S and $h = \mathsf{height}(G)$. The G-representation of k can be computed in $O(\log(\ell) + h)$ time by first doing a binary search on the nonterminal edges in S, and then following the correct nonterminal edges in the right-hand sides of rules until the node is reached. (2) Given the G-representation $e_0 e_1 \cdots e_n v$, the incoming neighbors are computed as follows. We return every internal node u in $\mathsf{rhs}(A_n)$ that has an edge to v. For every external node u in $\mathsf{rhs}(A_n)$ that has an edge to v, we need to compute its node ID, which is done by calling the method $\mathrm{getID}(e_0 e_1 \cdots e_n u)$. Clearly, this is done in $O(h)$ time. For every nonterminal edge e in $\mathsf{rhs}(A_n)$ that has an attachment to v, i.e., v appears in $\mathsf{att}(e)$ at some position p, we compute the node IDs of all terminal nodes produced by A (the label of e) and having an edge to the pth external node. We use a recursive method $\mathrm{getNeighboring}(e, p)$ which returns the set of node IDs that are neighbors to the pth external node within the subgraph derived from e. To do so, iterate over the edges incident with the pth external node. Let e' be the current edge. If e' is terminal and attached to an internal node, add the internal nodes ID to the result set. If it is terminal and attached to another external node w, we add the ID obtained by $\mathrm{getID}(e_0 e_1 \cdots e_n e w)$ to

the result set. If the edge is nonterminal, let p' be the position of the pth external node in $\mathsf{att}(e')$ and add the result of getNeighboring(e', p') to the result set. The runtime is $O(h)$ per node, and thus a total of $O(nh)$, where n is the number of neighbors.

Proposition 4. *Let G be an SL-HR grammar and k a node ID, i.e., $k \in \{1, \ldots, |\mathsf{val}(G)|_V\}$. Let n be the number of in (or out) neighbors of k in $\mathsf{val}(G)$. The node IDs of these n nodes can be computed in time $O(\log(\ell) + nh)$ where ℓ is the number of nonterminal edges in S and $h = \mathsf{height}(G)$.*

Note that for string grammars, data structures have been presented that guarantee *constant time* per move from one letter to the next (or previous) [31]. This result has been extended to grammar-compressed trees [32], and we expect it can be generalized to SL-HR grammars.

Speed-Up Queries: One attractive feature of straight-line context-free grammars is the ability to execute finite automata over them without prior decompression. This was first proved for strings (see [14]) and was later extended to trees (and various models of tree automata, see [33]). The idea is to run the automaton in one pass, bottom-up, through the grammar. As an example, consider the grammar $S \to AAA$ and $A \to ab$ from the Introduction, and an automaton $\mathcal{A}$ that accepts strings (over $\{a, b\}$) with an odd number of a's. Thus, $\mathcal{A}$ has states q_0, q_1 (where q_0 is initial and q_1 is final) and the transitions (q_0, a, q_1), (q_1, a, q_0), and (q, b, q) for $q \in \{q_0, q_1\}$. Since the actual active states are not known during the bottom-up run through the grammar, we need to run the automaton *in every possible state* over a rule. For the nonterminal A we obtain (q_0, A, q_1) and (q_1, A, q_0), i.e., running in state q_0 over the string produced by A brings us to state q_1, and starting in q_1 brings us to q_0. Since S is the start nonterminal, we are only interested in starting the automaton in its initial state q_0. We obtain the run $(q_0, A, q_1)(q_1, A, q_0)(q_0, A, q_1)$, i.e., the automaton arrives in its final state q_1 and hence the grammar represents a string with odd number of a's. It should be clear that the running time of this process is $O(|Q||G|)$, where Q is the set of states of the automaton, and G is the grammar.

Unfortunately, for graphs there does not exists an accepted notion of finite-state automaton. Nevertheless, properties that can be checked in one pass through the derivation tree of a graph grammar have been studied under various names: "compatible", "finite", and "inductive", and it was later shown that these notions are essentially equivalent [34]. Courcelle and Mosbah [35] show that all properties definable in "counting monadic second-order logic" (CMSO) belong to this class, and by their Proposition 3.1, the complexity of evaluating a CMSO property ψ over a derivation tree t can be done in $O(|t|\eta)$, where η is an upper bound on the complexity of evaluation on each right-hand side of the rules in t. How can we apply this result to SL-HR grammars G? Let $G = (N, S, P)$ so that every right-hand side in P contains at most two nonterminal edges. We are interested in *data complexity*, i.e., we assume ψ to be fixed. Note that the size of the derivation tree t of G can be exponential in $|G|$. It is thus prohibitive to construct t, and instead their proposition above needs to be generalized to dags (directed acyclic graphs) d that represent t. Next, we eliminate η as follows. We convert G into a new SL-HR grammar G' so that G' is in Chomsky normal form, i.e., every right-hand side (including the start graph) has at most two edges (see, e.g., Proposition 3.13 of [17]). The derivation dag of G' now has size $O(|G|)$. Let m be the maximum of the rank of G and of $|S|_V$. In the worst-case the start graph of G' has one nonterminal edge that is incident with all nodes in S. Thus the maximal size of a right-hand side of G' is in $O(m)$.

Proposition 5. *Let ψ be a fixed CMSO property. For a given SL-HR grammar G it can be decided in $O(|G|m)$ time whether or not ψ holds on $\mathsf{val}(G)$, where m is the maximal size of a rule of G.*

Note that Proposition 5 is often stated under a fixed tree decomposition t of width k of the graph g and then simply becomes $O(|g|)$. The CMSO (or compatible or finite) graph properties have been extended to functions from graphs to natural numbers, see e.g., Section 5 of [35]. They can be evaluated with a similar complexity as in Proposition 5. For the same explanation as above, this result can be applied to SL-HR grammars. Without stating this result explicitly, we mention some of the well-known CMSO functions: (1) maximal and minimal degree, (2) number of connected components, (3) number of simple cycles, (4) number of simple paths from a source to a target, and (5) maximal and minimal length of a simple cycle.

Reachability Queries: An important class of queries are *reachability queries*. For a given simple graph g and nodes u and v such a query asks if v is reachable from u, i.e., if there exists a path from u to v in g. It is well known that this problem can be solved in $O(g)$ time. How can we solve this problem on an SL-HR grammar G? Certainly, (u, v)-reachability is CMSO definable and therefore Proposition 5 gives us an upper bound of $O(|G|^2)$. We now give a direct *linear time* algorithm.

Theorem 6. *Let g be a simple graph and $G = (N, S, P)$ an SL-HR grammar with $\mathsf{val}(G) = g$. Given nodes $u, v \in V_g$, it can be determined in $O(|G|)$ time whether or not v is reachable from u in g.*

Proof: We first compute G-representations u', v' of u and v in $O(|G|)$ time, as described in Section V. We traverse G bottom-up with respect to $\leq_{\mathsf{NT}}$ in one pass and compute for every nonterminal A its *skeleton graph* $\mathsf{sk}(A)$. The set of nodes of $\mathsf{sk}(A)$ is given by the external nodes $\{v_1, \ldots, v_n\}$ of the right-hand side h of A. The edges are computed as follows. First, assume that h is a terminal graph. We determine the strongly connected components in h in linear time (e.g., using Tarjan's algorithm [36]). Let h' be the corresponding graph which has as nodes the strongly connected components of h. We remove from h' each strongly connected component C that does not contain external nodes. This is done by inserting for every pair of edges e_1, e_2 such that e_1 is an edge from a component $D \neq C$ into C and e_2 is an edge from C to a component $E \neq C$ (with $E \neq D$), an edge from component D to component E. Finally, we replace each component by a cycle of the external nodes of that component, and, for an edge from a component D to a component E we add an edge from an arbitrary external node of D to one of E. After having computed in $O(|G|)$ time the skeleta for all nonterminals, we can solve a reachability query as follows. Let S' be the graph obtained from S by replacing each nonterminal edge by its skeleton graph; clearly, it can be obtained in $O(|G|)$ time.

Case 1: Assume that u' and v' are of the form u'' and v'', i.e., both nodes are in the start graph. It should be clear that v is reachable from u in $\mathsf{val}(G)$ if and only if v'' is reachable from u'' in S'. The latter is checked in $O(|S'|)$ time.

Case 2: Let $u' = e_0 \cdots e_n u''$ and $v' = f_0 \cdots f_m v''$. Let A_i be the label of e_{i-1} for $i \in \{1, \ldots, n\}$ and let B_j be the label of f_{j-1} for $j \in \{1, \ldots, m\}$. We determine the set E_n of external nodes of the right-hand side h_n of A_n that are reachable from u'' in h_n. This is done by replacing the (at most two) nonterminal edges in h_n by their skeleton graphs, and then running a standard reachability test. We now move up the derivation tree (viz. to the left in u'), at each step computing a subset E_i of the external nodes of $\mathsf{sk}(A_i)$: we locate the nodes corresponding E_{i+1} in $\mathsf{sk}(A_i)$ and determine the set E_i of external nodes reachable from these. Finally, we obtain a set E_0 of nodes in S' (all incident with the edge e_0). In a similar way we compute a set F_0 of nodes in S that are incident with f_0 (and can reach v'). Finally, we check if a node in F_0 is reachable from a node in E_0. This is done by adding edges over F_0 that form a cycle, and edges over E_0 that form a cycle. We now pick arbitrary nodes f in F_0 and e in E_0 and check if f is reachable from e in S'. ■

VI. Conclusions

We presented gRePair, a compressor based on straight-line hyperedge replacement grammars. It is a direct generalization of previous RePair algorithms for string- and tree-based data. Note that gRePair over string- and tree-graphs obtains similar compression ratios as the original specialized versions for strings and trees [15], [16]. On our datasets of RDF and version graphs, gRePair produces the smallest representations we are aware of. We proved that reachability queries can be solved in linear time with respect to the grammar, thus offering speed-ups proportional to the compression. In the future we want to find more query classes with this property (e.g., regular path queries), implement such query evaluation and compare its performance with state-of-the-art systems. We would like to investigate other node orderings that improve compression, and design better algorithms for approximating maximum non-overlapping sets of digram occurrences. We would like to apply our compression as index structures of graph databases.

References

[1] W. Fan, X. Wang, and Y. Wu, "Querying big graphs within bounded resources," in *SIGMOD*, 2014, pp. 301–312.

[2] W. Han, J. Lee, and J. Lee, "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *SIGMOD*, 2013, pp. 337–348.

[3] Y. Cao, W. Fan, J. Huai, and R. Huang, "Making pattern queries bounded in big graphs," in *ICDE*, 2015, pp. 161–172.

[4] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *ICDE*, 2014, pp. 568–579.

[5] W. Fan, "Graph pattern matching revised for social network analysis," in *ICDT*, 2012, pp. 8–21.

[6] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: mining peta-scale graphs," *Knowl. Inf. Syst.*, pp. 303–325, 2011.

[7] U. Kang, D. H. Chau, and C. Faloutsos, "Pegasus: Mining billion-scale graphs in the cloud," in *ICASSP*, 2012, pp. 5341–5344.

[8] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, and G. Navarro, "Compressed vertical partitioning for efficient RDF management," *Knowl. Inf. Syst.*, pp. 439–474, 2015.

[9] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *SIGMOD*, 2012, pp. 157–168.

[10] P. Boldi and S. Vigna, "The webgraph framework I: compression techniques," in *WWW*, 2004, pp. 595–602.

[11] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *SIGKDD*, 2009, pp. 219–228.

[12] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," *IEEE Transactions on Information Theory*, pp. 2554–2576, 2005.

[13] A. Amir, G. Benson, and M. Farach, "Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files," *J. Comput. Syst. Sci.*, pp. 299–307, 1996.

[14] M. Lohrey, "Algorithmics on SLP-compressed strings: A survey," *Groups Complexity Cryptology*, pp. 241–299, 2012.

[15] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proceedings of the IEEE*, pp. 1722–1732, 2000.

[16] M. Lohrey, S. Maneth, and R. Mennicke, "XML tree structure compression using RePair," *Information Systems*, pp. 1150–1167, 2013.

[17] J. Engelfriet, "Context-free graph grammars," in *Handbook of Formal Languages, Vol. 3*, G. Rozenberg and A. Salomaa, Eds., 1997, pp. 125–213.

[18] L. Peshkin, "Structure induction by lossless graph compression," in *DCC*, 2007, pp. 53–62.

[19] F. Claude and G. Navarro, "Fast and Compact Web Graph Representations," *ACM Trans. Web*, pp. 16:1–16:31, 2010.

[20] S. Grabowski and W. Bieniecki, "Tight and simple web graph compression for forward and reverse neighbor queries," *Discrete Applied Mathematics*, vol. 163, pp. 298–306, 2014.

[21] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of web graphs with extended functionality," *Inf. Syst.*, vol. 39, pp. 152–174, 2014.

[22] C. Hernández and G. Navarro, "Compressed representations for web and social graphs," *Knowl. Inf. Syst.*, pp. 279–313, 2014.

[23] G. Buehrer and K. Chellapilla, "A Scalable Pattern Mining Approach to Web Graph Compression with Communities," in *WSDM*, 2008, pp. 95–106.

[24] H. Liefke and D. Suciu, "XMILL: an efficient compressor for XML data," in *SIGMOD*, 2000, pp. 153–164.

[25] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in *VLDB*, 1997, pp. 436–445.

[26] M. Lohrey and S. Maneth, "The complexity of tree automata and XPath on grammar-compressed trees," *Theor. Comp. Sci.*, pp. 196–210, 2006.

[27] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, pp. 194–203, 1975.

[28] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto, "RDF compression: basic approaches," in *WWW*, 2010, pp. 1091–1092.

[29] M. A. Martínez-Prieto, J. D. Fernández, and R. Cánovas, "Compression of RDF dictionaries," in *SAC*, 2012, pp. 340–347.

[30] S. Maneth and T. Sebastian, "Fast and tiny structural self-indexes for XML," *CoRR*, vol. abs/1012.5696, 2010.

[31] L. Gasieniec, R. M. Kolpakov, I. Potapov, and P. Sant, "Real-time traversal in grammar-based compressed files," in *DCC*, 2005, p. 458.

[32] M. Lohrey, S. Maneth, and C. P. Reh, "Traversing grammar-compressed trees with constant delay," in *DCC*, 2016, to appear.

[33] M. Lohrey, S. Maneth, and M. Schmidt-Schauß, "Parameter reduction and automata evaluation for grammar-compressed trees," *J. Comput. Syst. Sci.*, pp. 1651–1669, 2012.

[34] A. Habel, H. Kreowski, and C. Lautemann, "A Comparison of Compatible, Finite, and Inductive Graph Properties," *Theor. Comput. Sci.*, pp. 145–168, 1993.

[35] B. Courcelle and M. Mosbah, "Monadic Second-Order Evaluations on Tree-Decomposable Graphs," *Theor. Comput. Sci.*, pp. 49–82, 1993.

[36] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.*, pp. 146–160, 1972.