Incremental Updates on Compressed XML

Incremental Updates on Compressed XML

Stefan Böttcher*, Rita Hartel*, Thomas Jacobs* and Sebastian Maneth†
*University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany
Emails: {stb@, rst@, tjacobs@mail.}uni-paderborn.de
†University of Edinburgh, Informatics Forum, Crichton Street, Edinburgh, EH8 9AB, UK
Email: smaneth@inf.ed.ac.uk

***Abstract*—XML tree structures can be effectively compressed using straight-line grammars. It has been an open problem how to update straight-line grammars, while keeping them compressed. Therefore, the best previous known methods resort to periodic decompression followed by compression from scratch. The decompression step is expensive, potentially with exponential running time. We present a method that avoids this expensive step. Our method recompresses the updated grammar directly, without prior decompression; it thus greatly outperforms the decompress-compress approach, in terms of both space and time. Our experiments show that the obtained grammars are similar or even smaller than those of the decompress-compress method.**

I. Introduction

Typical XML document trees compress well by means of dictionary-based methods. Buneman, Koch, and Grohe [1] show that XML document trees compress to about 10% of their edges, when represented as minimal DAGs (directed acyclic graphs). The idea of DAGs is to represent repeating subtrees only once. This idea was generalized to the sharing of repeated (connected) XML subgraphs by Busatto, Lohrey, and Maneth [2]. They show that *straight-line tree* (*SLT*) *grammars* compress typical XML trees to 3% of their edges. These ratios are further improved by the current state-of-the-art SLT compressor TreeRePair [3].

Dictionary-based compression methods are special, because they can provide in-memory representations which are queryable without decompression. TreeRePair provides the smallest queryable in-memory representation of XML document trees that we are aware of [4].

These features make SLT grammars ideal for in-memory XML processing. The downside however is: no previous method supports *efficient updates*, i.e., updates that can be executed on the grammar without decompression to the tree and that keep the updated grammar small. Updates lead to a significant blow-up in grammar size. In order to overcome this problem, the best known method is, after some updates, to decompress to a tree and then compress the resulting tree again. This can take exponential time and space. Therefore, SLT-compressed trees have only been applied in static scenarios. Many applications however rely on *dynamic* trees that change frequently; consider for instance DOM-trees used in web browsers.

This paper presents the first implementation of SLT-compressed trees with efficient updates. The idea is to generalize TreeRePair from running on trees to running on grammars. In this way, updates can be realized by first efficiently producing an updated grammar and then running GrammarRePair over this grammar. Our contributions are:

(1) We devise the compression algorithm GrammarRePair which executes RePair compression directly on an SLT grammar, without prior decompression.
(2) We show that on typical XML trees, GrammarRePair compresses as well as TreeRePair; on some trees it even compresses considerably better.
(3) We investigate the update performance of SLT grammars under GrammarRePair compression:
   a) For sequences of thousands of updates (inserts and deletes), our grammars achieve virtually the same compression ratios as achieved by the update-decompress-compress (udc) method.
   b) For typical files, our space consumption overhead is less than 1% compared to the compression result of udc, while naive updates (without further compression) cause about 40% overhead. For a few files with extreme compression ratios, our overhead compared to the compression result of udc is a factor of two, while naive updates cause blow-ups of more than 100-fold.
   c) For XML trees with more than 100k edges, our update time is faster than that of udc. For trees with more than 200k edges, our update time is even faster than the mere compression time of udc.

Let us describe in a few more words the idea of our method. The idea of RePair compression is to repeatedly replace the most frequent *digram* by a new nonterminal. For strings [5] a digram simply consists of two adjacent symbols. Thus, on the string $w = abababa$ we can replace ab by a nonterminal symbol A, obtaining $AAAa$, and then replace AA by B, obtaining the final grammar $G_w = \{S \to BBa, B \to AA, A \to ab\}$. Note that the size of this grammar (= sum of lengths of right-hand sides) is 7, while the original string has length 9. In a tree, a digram is a triple (a, i, b) where a, b are labels of adjacent nodes and i is a child number. Note that a naive implementation that counts digrams after each round of replacements, does *not* run in linear time. A linear time implementation is non-trivial. It was solved by Larson and Moffat [5] through careful incremental updates of digram occurrence lists together with clever data structures such as a priority queue of length $\sqrt{n}$ (where n is the length of the input string) holding frequent digrams. To see the challenge caused by an update, consider changing the w above into $v = bw$. A grammar for this is $H_v = \{S \to bBBa, B \to AA, A \to ab\}$. However, this is *not* the grammar RePair would produce on v, because initially now ba is the most frequent digram. The desired smaller grammar would be $G_v = \{S \to BBA, B \to AA, A \to ba\}$. The question arises, how this grammar G_v can efficiently be obtained from the grammar H_v?

In our setting, we need to count digram occurrences not only in a string, but in a tree T generated by the SLT grammar G, but without decompressing G to T. We show how this is possible in one pass over an SLT grammar.

The main challenge is the efficient replacement of all occurrences of a given digram $\alpha = (a, i, b)$ in a tree T generated by a grammar G, in one pass through G, and with minimal decompression of G. One major technical issue is that an occurrence of α need not reside in just one rule, but can span over several rules. This implies that we first have to apply these rules (thus locally decompressing) in order to make this occurrence explicit to be able to replace it. The delicate issue is to do as few rule applications as possible (by running them in a particular order), and to apply these rules in a "compact" way (by introducing new rules). At the end of the Preliminaries, we give a more technical outline of our technique.

Related Work

The idea of updates through path isolation is used by Fisher and Maneth [6]. For typical XML documents, they find that long update sequences of delete/inserts (on the same document) increase grammar sizes by around 40%. Their SLT grammars are produced by BPLEX [2].

Bätz, Böttcher, and Hartel [7] use path isolation to find that updates can be performed faster than using the decompress-update-compress method. Moreover, they show how to perform multiple updates in parallel during one grammar pass. This approach is further improved by Böttcher, Hartel, and Jacobs [8] by computing DAG-compressed representations of the update positions induced by an XPath query (when evaluated over the given SLT grammar).

For straight-line string grammars, digram replacement in a given grammar has been used by Jez [9] in his recompression framework. For this, he distinguishes "non-crossing" digrams that fully reside in rules (and thus are easy to replace) from "crossing" digrams which are more difficult to replace. He shows that all occurrences of a digram can be replaced in linear time with respect to the size of the grammar. Jez's recompression approach has been generalized to trees by Jez and Lohrey [10] in order to present a compression algorithm by SLT grammars with the currently best proven approximation ratio (with respect to the minimal SLT grammar, which is NP-complete to compute, cf. [11]).

Succinct trees offer are a compact tree representation, see, e.g., the work by Munro and Raman [12]. They have been used to build in-memory DOM representations for XML trees by Delpratt, Raman, and Rahman [13]; while this representation is space efficient and supports fast navigation, it does not support updates. Dynamic succinct versions are more complicated and efficient implementations are still missing, even though appropriate data structures have recently been implemented by Joannou and Raman [14], cf. the concluding remarks in [15].

II. Preliminaries

Our formal models of an XML tree and a tree grammar follow and slightly extend [3]. Like TreeRePair, we consider an input document as a labeled binary tree, where each non-leaf node has exactly two children, i.e., it has *rank* two: its first-child and its next-sibling (Figure 1). Different from TreeRePair, we introduce a leaf node with label $\perp$ called the empty node to represent non-existing first-child or next-sibling nodes. Our compression works on ranked labeled ordered trees of which binary XML trees are just a special case.

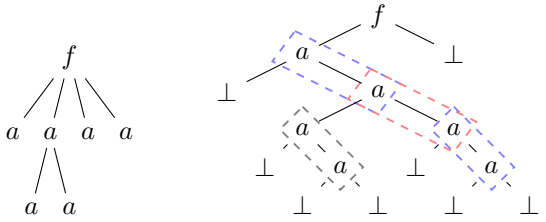


Fig. 1. An XML tree and its binary tree

A ranked alphabet Σ consists of a finite set of symbols each of which is associated a natural number, called the *rank* of the symbol. We fix the set $Y = \{y_1, y_2, \dots\}$ of *formal parameters*, and assume it to be disjoint with all other ranked alphabets. A finite tree (over Σ) with parameters is a tree t, such that for every node u of t, if the label of u is a symbol in Σ of rank k, then u has exactly k children, and, if the label of u is in Y, then u has zero children. Let *params*(t) denote the number of nodes of t that have their label in Y. We demand that if *params*$(t) = m$, then for each $i \in \{1, \dots, m\}$, there is exactly one node in t labeled by y_i. Thus, our trees are *linear in the parameters* $Y_m = \{y_1, \dots, y_m\}$.

Let t, t' be trees and v be a node of t. By $t[v/t']$, we denote the *replacement* of the subtree rooted at node v by the tree t'. Now let *params*$(t') = m$ and $t_1, \dots, t_m$ be trees. Then $t[y_1/t_1, \dots, y_m/t_m]$ denotes the tree obtained from t by replacing each occurrence of y_j by the tree t_j, for $1 \leq j \leq m$. If $t[v/t'[y_1/t_1, ..., y_n/t_n]] = t$, then t' is a *pattern of rank* m occurring in tree t at node v. For example, the pattern $a(y_1, a(y_2, y_3))$ occurs at four different nodes in the tree of Figure 1, namely, at the top a-node of each dashed box.

To compress a ranked labeled ordered tree by extracting patterns that occur multiple times, so-called *straight-line linear context-free (SLCF) tree grammars* can be used. Using the model of [3], a linear context-free tree grammar is a 4-tuple $G = (\mathcal{F}, \mathcal{N}, P, S)$, where $\mathcal{F}$ is the set of ranked terminal symbols (including the null pointer $\perp$ with *rank*$(\perp) = 0$), $\mathcal{N}$ is the set of ranked nonterminal symbols, $\mathcal{F} \cap \mathcal{N} = \emptyset$, P denotes the set of rules, and $S \in \mathcal{N}$ is the start nonterminal symbol not occurring on the right-hand side of any rule. For each $R \in \mathcal{N}$ of rank m, there is exactly one rule $(R \to t_R) \in P$, where t_R must not consist of a single node that is a parameter. Each such t_R is a tree over $\mathcal{F} \cup \mathcal{N} \cup Y_m$, such that each $y_j \in Y_m$ occurs exactly once in t_R. The derivation relation of G is denoted by $\Rightarrow_G$ (or by $\Rightarrow$). For trees t, t', $t \Rightarrow_G t'$ if (1) t contains a subtree $t_0 = A(t_1, \dots, t_m)$ with $A \in \mathcal{N}$ and (2) t' is obtained from t by replacing t_0 by the tree $t_A[y_1/t_1, \dots, y_m/t_m]$.

In order to identify a particular node v occurring in the right-hand side t_B of a grammar rule $B \to t_B$, we denote v as (B, m), where n is the number of nodes (terminals, nonterminals, parameters) occurring in t_B, $1 \leq m \leq n$, and (B, m) is the m-th node of t_B in preorder.

For example, consider the grammar $G = (\mathcal{F}, \mathcal{N}, P, S)$ with $\mathcal{F} = \{a, f, \perp\}$, $\mathcal{N} = \{A, B, S\}$, and $P = \{S \to$

$f(A(B,B),\bot), B \to A(\bot,\bot), A \to a(\bot, a(y_1,y_2))\}$. Then, $B \Rightarrow A(\bot,\bot)$; we apply the A-rule to the tree $A(\bot,\bot)$ by replacing y_1 and y_2 in t_A by $\bot$, i.e., $A(\bot,\bot) \Rightarrow a(\bot, a(\bot,\bot)) = t$. Then, $A(B,B) \Rightarrow a(\bot, a(B,B)) \Rightarrow a(\bot, a(t,t))$. Thus, $S \Rightarrow f(A(B,B),\bot)$ which derives to the binary tree $f(a(\bot, a(t,t)),\bot)$ which is shown in Figure 1.

Furthermore, $\mathit{ref}_G(Q) = \{(R,n) \mid (R \to t_R) \in P \wedge \mathit{label}(t_R,n) = Q\}$ denotes the set of the Q-labeled nonterminal nodes within the rules of G. If $(R,n) \in \mathit{ref}_G(Q)$, we also say that R *calls* Q, $\mathit{calls}_G(Q,R)$ for short. Then, the transitive closure, $\mathit{calls}^*_G(Q,R)$, contains all pairs of nodes, where R *directly* or *indirectly* calls Q. We consider only grammars that are non-recursive, also called *straight-line*, i.e., $\neg\exists Q : \mathit{calls}^*_G(Q,Q)$. We say that Q occurs before R in *anti-straight-line order (anti-SL)* in G, if $\mathit{calls}^*_G(Q,R)$.

To *inline* a rule, means for $(R,n) \in \mathit{ref}_G(Q)$ that occurs in the right-hand side of a rule $R \to t_R$, we modify tree t_R by replacing (R,n) by t_Q, where the i-th parameter node of t_Q is replaced by the subtree rooted in the i-th child node of (R,n). Formally, let $\mathit{rank}(Q) = m$ and let t_i be the i-th subtree of (R,n). That is, inlining Q into (R,n) is formally defined by the replacement $t_R[(R,n)/(t_Q[y_1/t_1,\ldots,y_m/t_m])]$. If afterwards $|\mathit{ref}_G(Q)| = 0$, we delete rule $Q \to t_Q$ from grammar G. For example, if we inline rule B for node $(S,3)$, we obtain $S \to f(A(A(\bot,\bot),B),\bot)$. Let $val_G(R)$ be the tree corresponding to R, i.e., the tree that we obtain when repetitively *inlining* nonterminals in t_R, until t_R does not contain any nonterminals. As a special case, $val_G(S) = T$ is the tree generated by the grammar G. Let b be a terminal node in any rule of a grammar G with start symbol S, and let $T = val_G(S)$ be the tree generated by inlining all rules into S. We define a *correspondence* between b and a set of nodes v in T by the following procedure. We mark b in G, and whenever a rule containing a marked node b is inlined, the mark is copied during the inlining step. When no more rules can be inlined, i.e., when we have transformed G into T, all the marked nodes in T are the nodes that *correspond* to b.

As defined in [3], a digram $\alpha = (a,i,b)$ is a triple, where $a,b \in \mathcal{F}$ and $i \in \{1,...,\mathit{rank}(a)\}$. It denotes an edge from an a-labeled node to its i-th b-labeled child node. An occurrence of a digram $\alpha = (a,i,b)$ is a pair of nodes v and w in T, such that $\mathit{label}(v) = a$ and v's i-th child is w with $\mathit{label}(w) = b$. Two occurrences of the same digram α are overlapping, if they share a common node in T. For two distinct occurrences of α, overlapping can only happen if $a = b$.

For example, in the tree T shown above in Figure 1, we have several occurrences of the digram $(a,2,a)$ marked by a dashed box. The occurrences marked with a blue box overlap with the occurrence marked by the red box.

Finally, let $\alpha = (a,i,b)$ be a digram with $\mathit{rank}(a) = m$ and $\mathit{rank}(b) = n$. The pattern t_X *representing* α is defined to be $a(y_1,\ldots,y_{i-1}, b(y_i,\ldots,y_{i+n-1}), y_{i+n},\ldots,y_{m+n-1})$.

α is called appropriate if

1) $\mathit{rank}(\alpha) = m+n-1 \le k_{in}$, where k_{in} is a predefined constant limiting the maximum numbers of parameters of a rule $X \to T_X$ and
2) α has more than one occurrence within T.

Outline of the Proposed Solution. Recompression of a grammar G is done stepwise by finding a most frequent digram $\alpha = (a,i,b)$ in val_G, partially decompressing G, such that all digram occurrences of α are isolated, and replacing all digram occurrences of α with a new nonterminal X. In order to minimize the amount of decompression during partial decompression of G, we first minimize the amount of inlining steps, i.e., replace only those nonterminals A by t_A, for which we can isolate a terminal a or a terminal b that belongs to a digram occurrence of α. Second, instead of replacing each nonterminal A determined in the first step by t_A, we transform t_A into an equivalent tree t'_A by recompressing parts of t_A that are not needed to isolate a or b, and then replace A by the smaller tree t'_A.

III. Grammar Updates and Motivating Examples

XML tree structures can be updated using three atomic update operations: renames, deletes, and inserts. For this section, we mostly consider *renames*. Let t be a binary tree representation of an XML tree, u a node of t, and σ be a label. Then $\mathsf{rename}(t,u,\sigma)$ denotes the tree obtained from t by relabeling the node u by the label σ. To guarantee that the resulting tree is again a correct binary tree representation of an XML tree, we require that $\sigma \neq \bot$ and that the label of u in t is not $\bot$. As an example, let $t = f(d(\bot, b(\bot, a(\bot, b(\bot,\bot)))))$ and let $u = 2$, i.e., u denotes the d-node in t. Then $\mathsf{rename}(t,u,a)$ is the tree $f(a(\bot, b(\bot, a(\bot, b(\bot,\bot)))))$.

A. Path Isolation

In order to perform an update on a tree that is given as grammar, we first need to "isolate" the node at which the operation is to be performed. To see this, consider the following string grammar G_8:

$$\{A \to BB, B \to CC, C \to DD, D \to ab\}.$$

This grammar represents the string $(ab)^8$. We would like to rename the first a-symbol in this string by a c. Clearly, if we simply change the a-symbol of the D-production into c, then the whole string changes to $(cb)^8$ which is not what we want. The issue is that D appears many (eight) times in the string, and only the very first occurrence should be changed. Thus, we must "unfold" the grammar until the particular terminal symbol is available, and then perform the change. In the example, we can use this derivation:

$$A \Rightarrow BB \Rightarrow CCB \Rightarrow DDCB \Rightarrow abDCB.$$

Since the symbol a occurs in its terminal form, at the correct position of the update, we may now change it, and obtain $cbDCB$. Thus, a grammar with rule $A \to cbDCB$ and the B, C, and D rules as before, represents the renamed tree. We call this process of making a node terminally available a "path isolation". As another example, consider this grammar G_{exp}:

$$\begin{array}{lcl} A & \to & A_1A_1 \\ A_i & \to & A_{i+1}A_{i+1}, \quad \text{for } 1 \le i \le 9 \\ A_{10} & \to & a. \end{array}$$

The grammar represents the string a^{1024}. Note that the size of the grammar is 21. Assume that we want to rename the letter at position 333 by the letter c. Since A_1 produces a string of length 512, we know that this position is produced by the left

A_1 in the A-production. Thus, we derive $A \Rightarrow A_2A_2A_1$. Since A_2 produces a string of length 256, we now need to expand the second A_2, to obtain $A_2A_3A_3A_1$. After a few more steps, we obtain

$$A_2A_4A_7A_8aA_{10}A_9A_6A_5A_3A_1$$

and as we may verify: $256 + 64 + 8 + 4 = 332$, thus, the a above is at the correct position.

Let G be a grammar and u a node in $val_G(S)$. In one pass through G, we can precompute for every nonterminal A of G of rank k, the numbers $\mathsf{size}(A, 0)$, $\mathsf{size}(A, 1)$, ..., $\mathsf{size}(A, k)$. These are the numbers of nodes in $val_G(A)$ that appear before y_1 (in pre-order), between y_1 and y_2, ..., after y_k. For instance, if $val_G(A) = f(y_1, g(h(a, y_2), g(a, y_3)))$ then $\mathsf{size}(A, 0) = 1$, $\mathsf{size}(A, 1) = 3$, $\mathsf{size}(A, 2) = 2$, and $\mathsf{size}(A, 3) = 0$.

Using these numbers, it is straightforward to determine a shortest derivation $S \Rightarrow^* \xi$, such that a terminal symbol occurs in ξ that uniquely represents u: we start at the root node of the right-hand side of S. Assume that it is a nonterminal X of rank k. Using the size-information, we can determine whether u is produced by X, and if not, determine the $j \in \{1, \ldots, k\}$ such that u is produced in the j-th subtree. The tree ξ is the *path isolation of G for u*, denoted by $\mathsf{iso}(G, u)$. Path isolation is used already in [6]. The following lemma is straightforward because each production is applied at most once during path isolation.

Lemma 1. Given a grammar G and a node u in $val_G(S)$, $|\mathsf{iso}(G, u)| \leq 2 \cdot |G|$.

B. Updating the Path-Isolated Grammar

As we have seen, even performing a single rename operation over a grammar can, due to path-isolation, cause a blow up of (almost) a factor two. Clearly, if we repeatedly apply more and more updates, then we can blow-up again and again, eventually loosing all compression of the grammar. In general this cannot be avoided: consider renaming every single node by a new unique symbol. Then the output cannot be compressed, thus, after n steps (where n is the size of the uncompressed tree) we obtain a grammar of size n, which does not exhibit any compression.

Typically, however, the grammars after update are *much larger* then they need to be. Consider again the grammar G_8 of before. We now insert a b symbol before the first symbol, and an a symbol after the last symbol. These positions are available immediately in the start rule's right-hand side, without any further path isolation needed. We obtain this grammar:

$$\{A \to bBBa, B \to CC, C \to DD, D \to ab\}.$$

The represented string is $b(ab)^8a$. How would RePair compress this string? Clearly, the most frequent digram now is ba and *not* ab as previously. Therefore, we want to execute repair compression directly on the grammar shown above, causing as little as possible decompression.

C. Grammar Recompression

Let us outline our algorithm on this example. At first, we need to count the number of occurrences of digrams. Let us defer this step until later, and assume that we have determined that ba is the most frequent digram and hence shall be replaced by the new nonterminal X. We traverse the productions bottom-up. Obviously, the right-hand side ab of D does not contain any occurrence of ba. We record that the *first letter* of D is an a, and that the *last letter* of D is a b. Consider now the production $C \to DD$. None of the D's contains ba, however, the string of DD does contain an occurrence, stemming from the last letter of the first D and the first letter of the second D. Hence, this production is changed into

$$C \quad \to \quad aXb.$$

With this, the right-hand side of B rewrites to $aXbaXb$ and hence becomes $aXXXb$. In order to limit decompression when we need to insert this string for B in the future, we would introduce a new auxiliary nonterminal Y for XXX. We refer to the memorization technique as *lemma generation*. Thus, we have $\{B \to aYb, Y \to XXX\}$ Finally, we replace B by aYb in the right-hand side of A to obtain: $baYbaYba$, which becomes $XYXYX$. Thus, we obtain the following grammar:

$$\{A \to XYXYX, Y \to XXX, X \to ba\}.$$

Note that the size of this grammar is 10, while the corresponding grammar *without* lemma generation has size 11 (its A rule has X^9 in its right-hand side). Thus, even in this tiny example, lemma generation is beneficial.

We now need to run a next round of RePair. Again, let us defer how digram occurrences are counted in this grammar, and assume that XX is the most frequent digram. We introduce $Z \to XX$ and change $Y \to XXX$ into $Y \to ZX$. With this, the right-hand side of A becomes $XZXXZXX$ and thus $XZZZZ$. The resulting grammar is $\{A \to XZZZZ, Z \to XX, X \to ba\}$ Even though this will not change the size of this grammar, RePair would replace ZZ by a new nonterminal W to yield this final grammar:

$$\{A \to XWW, W \to ZZ, Z \to XX, X \to ba\}.$$

Observe, when Y's right hand side XXX was changed to ZX. This is a *left-greedy* replacement. Such a replacement need *not* be optimal. Consider a rule $S \to XYYX$. Replacing Y by ZX in this rule yields $XZXZXX$ which contains only one occurrence of XX (and thus would become $XZXZZ$). Similarly, choosing the right-greedy XZ would give us $XXZXZX$ and then $ZZXZX$. To get an optimal result (especially in the presence of more occurrences of YX and XY), one would need to introduce two rules: $Y \to XZ$ and $Y' \to ZX$. With these rules, the S-rule becomes $XXZZXX$ and thus ZZZ. In the case of trees, even two rules would not be enough, e.g., the rule $Y \to a(a(y_1, y_2), a(a(y_3, y_4), y_5))$ would require four versions. In a tree, left-greedy matching of a digram is generalized to "top-down greedy" matching. In summary:

(1) we use *lemma generation* in order to keep the grammar compressed during replacement of digrams, and
(2) we always replace digrams in a *left-greedy* way, in order to only generate one version of a production.

IV. Grammar RePair Compression

TreeRePair [3] takes as input a labeled ordered binary tree T and produces as output an SLCF tree grammar G' with

$val_{G'}(S) = T$. Our generalization GrammarRePair takes as input an SLCF grammar G and produces a (smaller) grammar G', such that $val_{G'}(S) = val_G(S)$.

GrammarRePair works on so called digrams, i.e., pairs of adjacent terminal nodes of the tree T generated by G.

Algorithm 1, the steps of which are later described in more detail, shows an overview of GrammarRePair.

Algorithm 1: GRAMMARREPAIR($G = (F, N, P, S)$)

```
1 RETRIEVEOCCS(G)
2 while α = MOSTFREQUENTDIGRAM exists do
3     replace each occurrence of α in G by X, where X
      is a new nonterminal
4     P = P ∪ {X → t_X} with t_X representing α
5     F = F ∪ X
6     UPDATEDIGRAMOCCURRENCES
7 PRUNINGPHASE
```

First, GrammarRePair computes a set of digram occurrences within G for each digram and counts how often the occurrences are present within T (line 1). Afterwards, it iterates through the following steps:

1) A most frequent appropriate digram α regarding T is selected (line 2).
2) Each digram occurrence of α in G is replaced by X, where X is a new nonterminal (line 3).
3) A rule $X \to t_X$ is added to G where the pattern t_X represents the digram α (line 4). In the following steps, T is the tree generated by the grammar that treats X as a terminal (line 5).
4) As the grammar is modified, the sets of digram occurrences are updated (line 6).

Finally, if no more appropriate digram exists, the *Pruning Phase* (line 7) removes all rules that do not contribute to the compression. To be more precise, GrammarRePair consists of the following steps outlined in the sections IV-A to IV-D.

A. Counting and Locating Digram Occurrences

In this section, we first explain the steps of TreeRePair working on trees as a simple case, and then extend each step to work on arbitrary SLCF grammars.

Considering the simpler case of a tree as input, TreeRePair searches for digrams in a tree T and counts the sizes of maximal sets of non-overlapping occurrences of each digram. This is done in a bottom-up greedy way: TreeRePair traverses T in post-order and adds each occurrence of a digram α to the digram list of α, provided it does not overlap with an occurrence that is already stored in the digram list of α.

While the location of each digram occurrence (v, w) in T can be uniquely described by the child node w, as its parent v is unique, an SLCF grammar may contain a terminal node w that belongs to different digram occurrences $(v_1, w), (v_2, w)$.

For an arbitrary SLCF grammar G as input, we define digrams, digram occurrences, their generators, their tree parent, and their tree child in G as follows.

Let $R \to t_R$ be a rule in G and (R, n) a node in t_R with $n \neq 1$ and $label(t_R, n) \neq y_i, i \in \mathbb{N}$, i.e., (R, n) is neither the root of t_R nor a parameter. Then, (R, n) is called a *digram occurrence generator* of a *digram occurrence* $((A, k), (B, m))$ of a digram $\alpha = (a, i, b)$ in grammar G, where $a = label(t_A, k)$ and $b = label(t_B, m)$.

We call the node $(B, m) = \text{TREECHILD}(R, n)$ the *tree child* in G of this digram occurrence, and the node (A, k) with $((A, k), i) = \text{TREEPARENT}(R, n)$ the *tree parent* in G of this digram occurrence. Here, the functions TREECHILD and TREEPARENT are defined as follows.

Algorithm 2: TREECHILD(B,m)

```
1 while (B, m) is nonterminal do
2     B := label(t_B, m);
3     m := 1;
4 return (B, m)
```

The function TREEPARENT returns the tree parent of (R, n) and the child index i as follows. The auxiliary method $parent(t, m)$ computes the position of the parent node of the m-th node in preorder within t. Furthermore, $i = index(t, m)$ denotes that the m-th node within t is the i-th child node of the node at position $parent(t, m)$. Note that TREEPARENT is only called for nodes (A, k) that are not the root of the tree t_A, i.e., nodes with $k \neq 1$.

Algorithm 3: TREEPARENT(A,k)

```
1 while label(t_A, parent(t_A, k)) is nonterminal do
2     i := index(t_A, k);
3     A := label(t_A, parent(t_A, k));
4     k := index(t_A, y_i);
5 return ((A, parent(t_A, k)), index(t_A, k))
```

Consider the following fragment "Grammar 1" of an SLCF grammar

$$\begin{array}{lcl} C & \to & A(B(\bot), \bot) \\ A & \to & a(y_1, a(B(\bot), a(\bot, y_2))) \\ B & \to & b(y_1, \bot) \end{array}$$

For example, consider the node $(C, 2)$ labeled with the label B in the first rule of Grammar 1 as digram occurrence generator. By calling $\text{TREECHILD}(C, 2)$, we obtain the terminal $(B, 1)$ having label b as $(C, 2)$'s tree child. And by calling $\text{TREEPARENT}(C, 2)$, we obtain $((A, 1), 1)$, i.e., the terminal node $(A, 1)$ having label a as $(C, 2)$'s tree parent and the index 1 as the child index. This together forms an occurrence of the digram $\alpha = (a, 1, b)$.

Let (R, n) be a digram occurrence generator in G with tree child (B, m), tree parent (A, k), and child index i and let $T = val_G(S)$ be the tree generated by G. We define the set $CDO_{(R,n)}$ of digram occurrences in T *corresponding* to (R, n) as $CDO_{(R,n)} = \{(v_T, w_T) \mid v_T$ *corresponds to* (A, k) *and* w_T *corresponds to* (B, m) *and* w_T *is the* i*-th child of* v_T *in* $T\}$.

As one digram occurrence generator within a rule $Q \to t_Q$ of G can correspond to multiple digram occurrences within the tree T generated by G, we have to compute based on G, the number of digram occurrences within T. Informally, this depends on how often the nonterminal Q is used within G to generate T. Formally, we recursively define the function $usage_G$ with $usage_G(S) = 1$, and for each $Q \in \mathcal{N}\backslash\{S\}$: $usage_G(Q) = \sum_{(R,n)\in ref_G(Q)} usage_G(R)$. In general, for each digram occurrence generator (C, m_C), the number of corresponding digram occurrences within T is given by $usage_G(C)$.

For example, in Grammar 1, if we assume that $usage_G(C) = 3$ and A is called only twice in a further rule S of Grammar 1, $usage_G(A) = 2{*}usage_G(S) + usage_G(C) = 2 * 1 + 3 = 5$, i.e., the occurrence of the digram $\alpha = (a, 1, b)$ generated by node $(A, 4)$ corresponds to five occurrences of α in tree $val_G(S)$.

One major challenge when dealing with SLCF tree grammars instead of DAGs or trees is to find a maximum set of non-overlapping occurrences of a digram (b, i, b).

To avoid overlapping occurrences, we use the same restriction as TreeRePair uses on DAG-compressed grammars. That is, we do not consider occurrences of equal label digrams where the tree child is the root of a rule. Note however that we do consider occurrences of digrams of the form (b, i, b) that cross parameter boundaries.

Algorithm 4: RETRIEVEOCCS($G = (\mathcal{F}, \mathcal{N}, P, S)$)

```
1  foreach (C → t_C) ∈ P, C ∈ N,
2          in anti-SL order do
3      foreach (C, n) ∈ t_C in pre-order do
4          if n ≠ 1 ∧
5              label(t_C, n) ≠ y_j then
6              ((A, k), i) :=TREEPARENT(C,n);
7              (B, m) :=TREECHILD(C,n);
8              α := (label(t_A, k), i, label(t_B, m));
9              if (label(t_A, k) ≠ label(t_B, m))∨
10                     (label(t_C, n) ∉ N∧
11                     (A, k) ∉ occ_G(α)) then
12                 occ_G(α) := occ_G(α) ∪ {(C, n)};
```

Algorithm RETRIEVEOCCS computes the set $occ_G(\alpha)$ of generators of all non-overlapping occurrences of all digrams $\alpha = (b, i, b)$ and of all occurrences of digrams $\alpha = (a, i, b)$ within grammar G in parallel within a single pass through grammar G. Therefore, RETRIEVEOCCS assumes that the tree parents (A, k), the child indices i (line 6), and tree children (B, m) (line 7) of all occurrences have been precomputed in a prior single pass.

The algorithm works as follows: We traverse the set of rules in an anti-SL order. Each rule is traversed in a pre-order run which assures that we combine the digrams in a "top-down greedy" way. If the node (C, n) visited is not a formal parameter and if it is not the root of t_C, we check, whether a digram occurrence can be added for the digram occurrence generator (C, n). There are two cases where a digram occurrence generator can be added: first (line 9), its tree parent (A, k) and its tree child (B, m) have different labels, second, if $label(t_A, k) = label(t_B, m)$, the node (C, n) is no nonterminal, and $((A, k), (B, m))$ does not overlap with another occurrence of the same digram. Such an overlap can only happen, if the tree parent (A, k) is a tree child of another occurrence of the same digram α, so line 11 checks whether (A, k) was added to the set of occurrences of α before. The missing case – (C, n) is a nonterminal and $label(t_A, k) = label(t_B, m)$ – would create an overlapping occurrence crossing the root of a rule, i.e., we do not add this occurrence. This implies that our approach does not produce overlapping digram occurrences, and it does not add occurrences of equal label digrams that cross rule boundaries to the root of a rule. This allows us to directly replace digram occurrences where tree parent node and tree child occur within the same rule without preparing the grammar. However, as shown in the next section, occurrences covering nodes from different rules still need to be treated in a separate way.

Consider again the fragment Grammar 1 of an SLCF grammar. Table I shows the results of applying the inner loop (lines 3-12) of Algorithm RETRIEVEOCCS to the rule $A \to t_A$ of applying it to the rule $C \to t_C$. Note that we omitted all results having the empty tree $\perp$ as tree child. The column $\Delta occ_G(\alpha)$ contains the digram occurrence generators added to the set of digram occurrence generators. The nodes $(A, 1)$ and $(C, 1)$ calculated in the first lines of Table I and of Table II do not generate a digram occurrence, as the considered node is the root of each rule. Nodes $(A, 2)$ and $(A, 8)$ are parameters and therefore do not generate digram occurrences either. The digram occurrence generated by node $(A, 6)$ is not added to $occ_G(\alpha = (a, 2, a))$, as it overlaps with the previously stored occurrence generated by node $(A, 3)$. Node $(C, 2)$ is an example for a digram occurrence generator where the tree parent $(B, 1)$, the tree child $(A, 1)$, and the node generating the digram occurrence occur all within different rules.

node	tree parent	tree child	α	$\Delta occ_G(\alpha)$
$(A, 1)$	–	–	–	–
$(A, 2)$	–	–	–	–
$(A, 3)$	$((A, 1), 2)$	$(A, 3)$	$(a, 2, a)$	$(A, 3)$
$(A, 4)$	$((A, 3), 1)$	$(B, 1)$	$(a, 1, b)$	$(A, 4)$
$(A, 6)$	$((A, 3), 2)$	$(A, 6)$	$(a, 2, a)$	–
$(A, 8)$	–	–	–	–

TABLE I. RETRIEVEOCCS FOR RULE $A \to t_A$

node	tree parent	tree child	α	$\Delta occ_G(\alpha)$
$(C, 1)$	–	–	–	–
$(C, 2)$	$((A, 1), 1)$	$(B, 1)$	$(a, 1, b)$	$(C, 2)$

TABLE II. RETRIEVEOCCS FOR RULE $C \to t_C$

B. Replacing Digrams

We start again with considering the simpler case of trees. After computing all digrams and their occurrences, a most frequent digram $\alpha = (a, i, b)$, with $rank(\alpha) \leq k_{in}$ is selected, where k_{in} is a predefined constant limiting the maximum rank of digrams that are replaced. For α, TreeRePair introduces a grammar rule $X \to t_X$ with t_X representing α.

Let $v.i$ be the subtree rooted in the i-th child node of a node v. For each occurrence of α consisting of tree parent v and tree child w, TreeRePair applies an operation that is inverse to inlining X, i.e., it replaces the subtree rooted in v by a subtree $X(v.1, \ldots, v.(i-1), w.1, \ldots, w.\mathit{rank}(b), v.(i+1), \ldots, v.\mathit{rank}(a))$.

In order to replace digrams $\alpha = (a, i, b)$ in arbitrary SLCF tree grammars, first, GrammarRePair generates the same rule $X \to t_X$ as TreeRePair does. Then, Algorithm 5 or an optimized version of it, Algorithm 6, is used for replacing all occurrences of the digram α.

Algorithm 5: REPLACINGALLOCCURRENCESOF(α)

```
1 DD_α = DEPENDENCYDAG(G = (F, N, P, S))
2 foreach (Q → t_Q) ∈ P in anti-SL order do
3     foreach node (Q, n) ∈ t_Q do
4         if (Q, n) ∈ DD_α and (Q,n) is a nonterminal R
          then
5             inline t_R for (Q, n)
6     replace each digram occurrence of α in t_Q by X as
      done in TreeRePair
```

Algorithm 5 consists of two parts:

1) The computation of a so called DependencyDAG DD_α for α (line 1), which contains all the rules that need to be considered when replacing the occurrences of α. This DependencyDAG contains all paths of grammar nodes (Q, n) which are nonterminals that have to be expanded when computing the tree parent or the tree child of a digram occurrence generator of α by Algorithm 2 or Algorithm 3.
2) A loop (lines 2-6) proceeding bottom-up through DD_α that does the digram replacements (line 6), preceded in some cases by inlining steps (lines 3-5).

The replacement of $\alpha = (a, i, b)$ traverses the grammar G bottom-up (lines 2-6), and for each production rule $Q \to t_Q$ of G, the following steps are executed:

1) For each node (Q, n) that is contained in DD_α and that is a nonterminal R, we inline t_R for R, thereby isolating a and b.
2) As now a and b are isolated, and b is the i-th child of a, we replace the digram occurrence $\alpha = (a, i, b)$ as in TreeRePair.

C. Updating the Context

The updates of the contexts are mainly the same, no matter whether we work on trees or on grammars – considering that we know for grammars how to compute the tree child and the tree parent of a digram occurrence.

For each occurrence that is replaced, we update the digram occurrences within the neighborhood. Conceptionally, this step results in the same result as repeating the step described in Section IV-A. But in order to avoid unnecessary re-counting, only the occurrences that overlap with an occurrence of the replaced digram $\alpha = (a, i, b)$ have to be adapted. If we consider a digram occurrence of α having tree parent v and tree child w, such that v is the j-th child node of its parent node p, with $\mathit{label}(p) = q$, we have to delete the occurrence consisting of tree parent p and tree child v of digram (q, j, a) and instead add an occurrence of (q, j, X). Similarly, for each child c of w with $\mathit{label}(c) = d$ being the k-th child of w, we delete the occurrence consisting of tree parent w and tree child c of digram (b, k, d) and instead add an occurrence of $(X, i + k - 1, d)$. Finally, the complete digram list of digram α has to be deleted.

The replacement step and the context update are repeated as long as a digram exists that has less than k_{in} parameters and occurs more than once.

D. Pruning

Finally, the resulting grammar is pruned, i.e., so-called unproductive grammar rules are removed. This step again is similar, no matter whether we work on trees or on arbitrary SLCF tree grammars. The productiveness of a rule $R \to t_R$ can be calculated by value $\mathit{sav}_G(R) = |\mathit{ref}_G(R)| * (\mathit{size}(t_R) - \mathit{rank}(R)) - \mathit{size}(t_R)$, where $\mathit{size}(t_R)$ is the number of edges of tree t_R [3]. If $\mathit{sav}_G(R) < 0$ holds for a rule R, this rule is unproductive and it is removed by inlining.

Note that the order of removing unproductive rules matters, as inlining a rule Q into a rule R changes the size of t_R, and therefore $\mathit{sav}_G(R)$ changes. We follow the greedy strategy used by TreeRePair. We first remove all rules Q with $|\mathit{ref}_G(Q)| = 1$ and then analyze the grammar in anti-SL order to find unproductive rules.

E. Optimized Digram Replacement Algorithm

Before we can explain the main optimization idea, i.e., the export of rule fragments to new rules, we present an example motivating that inlining has to consider multiple rule versions.

Consider $\alpha = (a, 1, b)$ and the following fragment of a grammar, "Grammar 2", and assume that A and C, but not B are called elsewhere in Grammar 2.

$$\begin{array}{lcl} C & \to & A(\bot, A(A(B, \bot), A(B, A(\bot, \bot)))) \\ A & \to & b(a(y_1, c(d(a(y_2, \bot), \bot), \bot)), \bot) \\ B & \to & b(\bot, \bot) \end{array}$$

The nodes $(C, 3)$, $(C, 4)$, $(C, 5)$, $(C, 7)$, $(C, 8)$, and $(C, 9)$ are digram occurrence generators of α. In order to replace all digrams α, we have to isolate the tree parent a and the tree child b of these digram occurrence generator by inlining.

Inlining t_B into nodes $(C, 5)$ and $(C, 8)$ is needed for later digram replacement and makes the rule $B \to t_B$ superfluous.

As the nonterminal A is called in different contexts, i.e., different parts of t_A have to be isolated to isolate a or b, we consider different versions of the rule $A \to t_A$ and inline that version of t_A that matches the demands of the context.

Whenever a node with nonterminal A is a digram occurrence generator (e.g. nodes $(C, 3)$, $(C, 4)$, $(C, 7)$, and $(C, 9)$), we have to isolate t_A's root node $(A, 1)$.

Similarly, we have to isolate the tree parents: e.g. the node $(C, 1)$ has as second parameter a digram occurrence generator, i.e., $\mathit{val}_G((C, 1))$ *contains* a tree parent. Therefore, to isolate

the tree parent found in $val_G((C,1))$, we have to isolate node $(A,6)$ which is the parent of the second parameter within t_A before inlining t_A for $(C,1)$.

But this is not the only version of t_A that needs to be inlined. Nodes $(C,3)$ and $(C,7)$ contain the tree parents of both of their parameters (and are digram occurrence generators themselves), i.e., we need a further version of t_A, where we isolate the nodes $(A,1)$, $(A,2)$, and $(A,6)$. A third version of t_A is needed for inlining the node $(C,4)$, which contains the tree parent of its first parameter (and is a digram occurrence generator itself). Finally, a fourth version of t_A is needed for inlining into the node $(C,9)$ that does contain no tree parent, but is a digram occurrence generator itself.

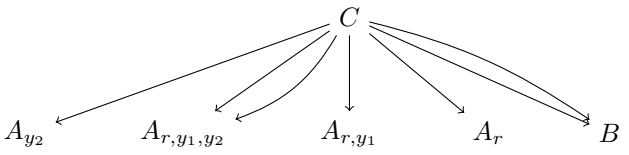
If we want to isolate the root node of t_A only, the first-child of t_A's root can be exported into a new rule to minimize the decompression when inlining t_A. I.e., instead of inlining t_A for $(C,9)$, in an optimized version, we create a rule version $A \to t_{A_r}$ and inline t_{A_r} for $(C,9)$ with $t_{A_r} = b(A^r(y_1,y_2),\perp)$ and generate a new export rule $A^r \to a(y_1, c(d(a(y_2,\perp),\perp),\perp))$.

On the other hand, if we have to isolate the root node and the parent of both parameters, the only fragment that is not needed in order to isolate the tree parent or the tree child is the subtree $c(d(y_2,\perp),\perp)$. So here we create a different version of t_A and inline $t_{A_{r,y_1,y_2}}$ with $t_{A_{r,y_1,y_2}} = b(a(y_1, A^{r,y_1,y_2}(a(y_2,\perp))),\perp)$ instead and generate a new export rule $A^{r,y_1,y_2} \to c(d(y_2,\perp),\perp)$.

We create in a similar way versions $t_{A_{y_2}}$ for $(C,1)$ and $t_{A_{r,y_1}}$ for $(C,4)$. Altogether, as we want to keep the grammar small, we need to consider multiple versions, in this case $t_{A_{y_2}}$, $t_{A_{r,y_1,y_2}}$, $t_{A_{r,y_1}}$, and t_{A_r} of a rule $A \to t_A$ for the nonterminal A, depending on in which context A is called in t_C.

These different versions of a rule $A \to t_A$ will be further optimized by exporting fragments to new rules as described at the end of this section.

To avoid a full decompression, instead of the simple $DependencyDAG$ (line 1 of Algorithm 5), we use the so-called *ReplacementDAG* RD_α to collect and manage the different versions of a rule, e.g. $A \to t_A$. In order to compute the ReplacementDAG RD_α we go top-down through the grammar, and for each rule $C \to t_C$, we do the following: For each digram occurrence generator (C,n) in t_C, we generate two flags: The first is an r flag attached to (C,n), the second is a y_i flag attached to (C,n)'s parent P in t_C, such that (C,n) is the i-th child of P in t_C. After having set all flags in t_C, for each nonterminal D in t_C that has a non-empty set F_D of flags attached to it, RD_α contains a node D_{F_D} and an edge from C to D_{F_D}. Repeatedly, for each new node C_{F_C} of RD_α, we generate a copy $C_\alpha \to t_{C_\alpha}$ of $C \to t_C$ and assign flags to nodes of t_{C_α} as follows. If $r \in F_C$ and $(C_\alpha,1)$ is a nonterminal R, we add a flag r to R. For each $y_i \in F_C$, we find the parent P of y_i in t_{C_α}, such that y_i is the k-th parameter of P, and if P is a nonterminal, we assign the flag y_k to P. After having set all flags in t_{C_α}, again for each nonterminal node R in t_{C_α} that has a non-empty set F_R of flags attached to it, we generate a node R_{F_R} in RD_α if it does not exist, and we add an edge from C_{F_C} to R_{F_R}. If we consider Example 2, we obtain the following ReplacementDAG RD_α:



Algorithm 6: REPLACINGALLOCCURRENCESOF(α)

```
1 RD_α = REPLACEMENTDAG(G = (F, N, P, S))
2 foreach node C_{F_C} of RD_α in bottom-up order do
3     APPLYINLININGSTEPS(C_{F_C})
4     replace each digram occurrence of α in t_Q by X as
      done in TreeRePair
5     EXPORTFRAGMENTSTONEWRULES(t_C)
6 Add exported rules to G
```

We replace Algorithm 5 by the optimized version, Algorithm 6, that uses the *ReplacementDAG* RD_α instead of the simple DependencyDAG to keep track of all paths to nonterminals that shall be expanded.

Algorithm 7: APPLYINLININGSTEPS(Q_{F_Q})

```
1  foreach node (Q, n) in occ_G(α) do
2      if (Q, n) is a nonterminal R then
3          inline t_R for (Q, n)
4      if parent(Q, n) is a nonterminal P then
5          inline t_P for parent(Q, n)
6  if r ∈ F_Q then
7      if root(t_Q) is a nonterminal R then
8          inline t_R for root(t_Q)
9          mark root(t_Q)
10 foreach parameter y_i in t_Q with y_i ∈ F_Q do
11     if parent(t_Q, y_i) is a nonterminal P then
12         inline t_P for parent(t_Q, y_i)
13         mark parent(t_Q, y_i)
```

The inlining steps of Algorithm 5, e.g., inlining a tree t_Q for a nonterminal Q, have the purpose to isolate the a node or the b node of a given digram occurrence $\alpha = (a,i,b)$. However, t_Q contains more nodes than the a or the b node to be isolated. In order to distinguish nodes that shall be isolated like the a or the b node of a digram occurrence $\alpha = (a,i,b)$ from other nodes of t_Q, Algorithm 7 not only applies inlining steps (lines 3, 5, 8, and 12), but also marks the nodes that shall be isolated (lines 9 and 13).

The key idea of the optimization called in line 5 of Algorithm 6 is the following. A tree t_Q that is inlined for multiple occurrences of Q may contain a connected fragment f_U of nodes all of which are non-marked, i.e., do not need to be considered for replacing occurrences of α. By inlining t_Q multiple times, multiple copies of f_U will occur. To reduce the grammar size, a new rule $U \to f_U$ will be inserted, and all occurrences of f_U can be replaced by U.

Assume, that we look for digrams $(a,1,b)$ and have a rule

$A \to t_A^0$ with $t_A^0 = a(y_1, d(e(\bot, \bot), a(y_2, \bot)))$. Then, $t_C = d(e(\bot, \bot), y_1)$ is such a connected fragment in t_A^0. By adding a rule $C \to t_C$, we can rewrite the rule $A \to t_A^0$ to its simpler form $A \to t_A$ with $t_A = a(y_1, C(a(y_2, \bot)))$.

When multiple occurrences of A have to be replaced by inlining, we can hope to get smaller grammars by inlining t_A for A instead of t_A^0 for A, as t_A is smaller than t_A^0.

Algorithm 8: EXPORTFRAGMENTSTONEWRULES(t_Q)

```
1 if |ref_Q(G)| > 1 and at least one node of t_Q is marked then
2     Identify in t_Q all root nodes r_U of a fragment f_U
      containing multiple connected non-marked
      non-parameter nodes.
3     foreach such r_U do
4         Create a tree t_U, such that t_U is a copy of the
          subtree of t_Q rooted in r_U
5     foreach t_U do
6         Substitute each subtree rooted in a marked node
          with a new parameter y_j
7         Add a rule R_U → t_U
8     Replace each fragment f_U in t_Q by applying the
      compression rule R_U → t_U to f_U
9 Remove all marks from t_Q
```

If a node is *marked* within the modified tree t_Q, we know that $Q \to t_Q$ will be inlined when continuing the bottom-up traversal of RD_α. However, only marked nodes need to be inlined or are needed in order to execute replacements of occurrences of α. Therefore, the optimization algorithm, Algorithm 8, in line 1, checks whether $|ref_G(Q)| > 1$ and at least one node of t_Q is marked, and if so, decreases the size of t_Q as follows: Each marked node of t_Q partitions t_Q into *fragments* of multiple connected non-marked, non-parameter nodes (line 2). In the previous example, d and e form such a fragment that is exported into the rule $C \to t_C$, where the parameter y_1 marks the position of the next marked node a of t_Q. For each such fragment f_U of t_Q that consists of multiple nodes, we generate a subtree t_U for f_U (lines 3-7), *export* f_U into a new rule $R_U \to t_U$ (line 7), and replace f_U by applying this rule (line 8), before we inline $Q \to t_Q$ itself.

Intuitively, exporting f_U of t_Q is inverse to inlining $R_U \to t_U$ in t_Q. *Exporting* a fragment f_U that is rooted in a node v_U of t_Q means that in t_Q, we replace the subtree t_{v_U} rooted in v_U with the tree $U(t_1, ..., t_n)$, such that $t_1, ..., t_n$ are the subtrees of t_{v_U} rooted in the top-most nodes of t_{v_U} in preorder that are not contained in f_U. Formally, $t_Q := t_Q[v_U/U(t_1, ..., t_n)]$. Then, t_U is $t_{v_U}[t_1/y_1, ..., t_n/y_n]$.

Finally, we unmark all marked nodes within the modified, but possibly smaller tree t_Q (line 9).

F. Concluding Example

In this example, we discuss the optimized replacement of the digram $\alpha = (a, 1, b)$ by a new rule $X \to a(b(y_1, y_2), y_3)$ on the grammar fragment Grammar 1. We assume that it is a fragment only, i.e., the rules A, B, and C are called by further nonterminals anywhere else in the grammar. For this fragment, the two occurrence generators of α are the nodes $(A, 4)$ and $(C, 2)$. The ReplacementDAG RD_α consists of the nodes $\{C, A_r, B\}$ and the edges $\{(C, A_r), (C, B), (A_r, B)\}$.

We start bottom-up with rule B. The root node $(B, 1)$ is marked, but there is no fragment to export, so rule B stays unchanged.

Next, we continue with rule A. First, we inline t_B for $(A, 4)$ yielding the new right-hand side $a(y_1, a(b(\bot, \bot), a(\bot, y_2)))$. Now, we can replace the digram occurrence by nonterminal X yielding $a(y_1, X(\bot, \bot, a(\bot, y_2)))$. The root node a is marked, and as we assume that A is called from further rules, we apply the optimization and export the fragment $X(\bot, \bot, a(\bot, y_2))$ into the new rule $D \to X(\bot, \bot, a(\bot, y_2))$. This leads to the new rule $A \to a(y_1, D(y_2))$.

Finally, we continue with rule C. We start with inlining the right-hand sides of rules A and B yielding the new right-hand side $a(b(\bot, \bot), D(\bot))$. Now, we replace the digram occurrence by X yielding $C \to X(\bot, \bot, D(\bot))$.

This leads to the new grammar version

$$\begin{array}{lcl} C & \to & X(\bot, \bot, D(\bot)) \\ D & \to & X(\bot, \bot, a(\bot, y_2)) \\ X & \to & a(b(y_1, y_2), y_3) \end{array}$$

V. EXPERIMENTS

We present an experimental evaluation of our prototype implementation of GrammarRePair. There are two groups of experiments: (1) Static Compression and (2) Dynamic Compression. In the static part, we gauge the compression, and memory behavior of GrammarRePair; these experiments are over static XML document trees, without any updates performed. In the dynamic part, we investigate the compression and runtime behavior of GrammarRePair for XML document trees under updates.

A. Setup

Machine. All tests are performed on an Intel Core2 Duo CPU P8400 @3GHz with 4GB of RAM running the 64 bit version of Linux 3.11.10-25. The tests of our prototype are performed using Java 1.9 (64 bit) with 3400MB heap space and a thread stack size of 4MB. The TreeRePair implementation is Version 20100428[1] and was compiled using g++ 4.8.1. All runtimes are measured using the user runtime given through the linux command 'time'. When measuring running times, we always perform four consecutive runs and then report the average over those runs.

Datasets. We execute over XML files that consist of element nodes only; thus, there is no text content of attributes (or other features of XML such as comments or processing instructions). These files have been obtained from well known benchmark files, by stripping off all non-element content. Almost all of our files can be found on the XMLComp-Bench [16] site, which provides the original XML documents, the structure-only documents (as used in the experiments),

[1] Available at https://code.google.com/p/\trp/downloads/detail?name=TreeRePair_20100428.zip

and the references to where these files originated. The two additional files used are Medline[2] and NCBI[3]. Table III lists all the files used in our tests, together with some document statistics: the number of edges and the depth (denoted "dp") of the original document tree. The number c-edges refers to the number of edges in the grammar obtained by GrammarRePair (calculated as defined in Section II); the last column shows the corresponding compression ratio of c-edges divided by #edges.

dataset	#edges	dp	c-edges	ratio (%)
EXI-Weblog	93434	2	42	0.04
XMark	167864	11	22105	13.17
EXI-Telecomp	177633	6	107	0.06
Treebank	2437665	35	503830	20.67
Medline	2866079	6	118067	4.12
NCBI	3642224	3	59	<0.01

TABLE III. DOCUMENT STATISTICS AND GRAMMARREPAIR-COMPRESSION RESULTS

B. Static Compression

Recall that GrammarRePair takes an SLT grammar G as input, and produces as output a new SLT grammar obtained by running RePair-compression over the tree represented by the grammar G. Thus, since the original grammar G may consist of a single rule with a large tree as right-hand side, GrammarRePair can be seen as a tree compressor itself. It is therefore reasonable to compare the tree compression behavior of GrammarRePair with existing compressors. This is done in two parts: first the compression ratios, then the running times are compared. In a third part, we investigate the memory consumption of GrammarRePair. This is measured in terms of the sizes of the intermediate grammars.

Compression Ratio Comparison. We compare the compression ratios obtained by TreeRePair, and our new GrammarRePair applied to trees as well as to grammars. In nearly all cases, our algorithm GrammarRePair produces similar or better results than TreeRePair. Better results are obtained on documents with very strong compression, viz. very small ratios; for instance, on EXI-Weblog (which compresses to only 0.1% by TreeRePair), our compressor GrammarRePair is better than TreeRePair by almost 20%. But in general, all three approaches compress as good as the others. That is, there is hardly a difference in the absolute compression ratio reached by TreeRePair, or our new GrammarRePair applied to trees as well as to grammars.

Memory Consumption. During compression, GrammarRePair produces a number of intermediate grammars, each of them being the outcome of a digram replacement. We measure the memory consumption in terms of the sizes of these intermediate grammars. We call the fraction (max. size of intermediate grammar) divided by (size of final grammar) the *blow-up*. Our experiment starts with a grammar, and then runs GrammarRePair over the grammar. Figure 2 shows the blow-up. The number of edges within each XML file, the reached compression ratio, and the compression ratio at maximum blow-up are shown below the name. The worst blow-up is just over 2, while for many files, it is only around a few percent above 1. This is easily explained: files with extreme compression ratios such as NCBI and EXI-Weblog contain long lists that are compressed exponentially. Breaking such a list "open" during the run of GrammarRePair can quickly double the size of the intermediate grammar. However, that intermediate grammar is still minuscule compared to the original tree; this is seen by comparing the compression ratios of the grammars against the original tree. As we can see, for NCBI and EXI-Weblog, these values coincide, i.e., the difference in intermediate grammar size can hardly be measured when compared against the original tree size. The biggest difference here is on XMark, where the intermediate grammar has around 12% compression, while the compression by GrammarRePair is around 8%.

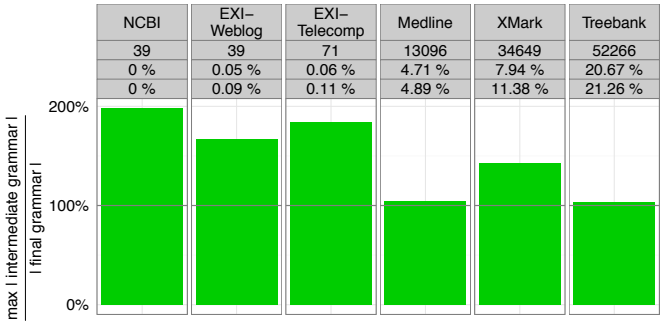


Fig. 2. Blow-up during recompression

Effect of the Optimization. In this experiment we compare the optimized version – i.e., exporting fragments t_U not needed in order to replace the digram into new rules $R_U \to t_U$ – to the non-optimized version.

For simplicity, we show string grammars here (to obtain a similar tree grammar, the reader may consider one additional root symbol, under which these grammars generate long children lists). For a natural number n, we consider the grammar G_n with these rules:

$$\begin{array}{lcll} S & \to & aA_nA_nb & \\ A_i & \to & A_{i-1}A_{i-1}, & \text{for } 1 \le i \le n \\ A_0 & \to & ba. & \end{array}$$

The grammar G_n produces a string consisting of a list of $n+1$ pairs of siblings a, b. This grammar is recompressed to the grammar with these rules:

$$\begin{array}{lcll} S & \to & B_0B_iB_i & \\ B_i & \to & B_{i-1}B_{i-1}, & \text{for } 1 \le i \le n \\ B_0 & \to & ab. & \end{array}$$

We choose values for n in the range from 64 to 4096. As this example compresses exponentially, i.e., the size of the tree $\mathsf{val}(G_n)$ is exponential in the size of the compressed grammars, the obtained grammars have at most 44 edges.

Figure 3 shows the results of our experiments. As we can see, if we use the optimization of Algorithm 8, the size of the grammar as well as the runtime seems to stay linear in the size of the compressed grammar. As expected, we reach a blow-up of less than 2 (the green curve varies from $1.2 - -1.7$). Due to the exponential compression of our files, we have several

[2]Available at http://www.ncbi.nlm.nih.gov/pubmed
[3]Available at http://snp.ims.u-tokyo.ac.jp

input XML documents of different sizes and thus several data points for each resulting grammar size.

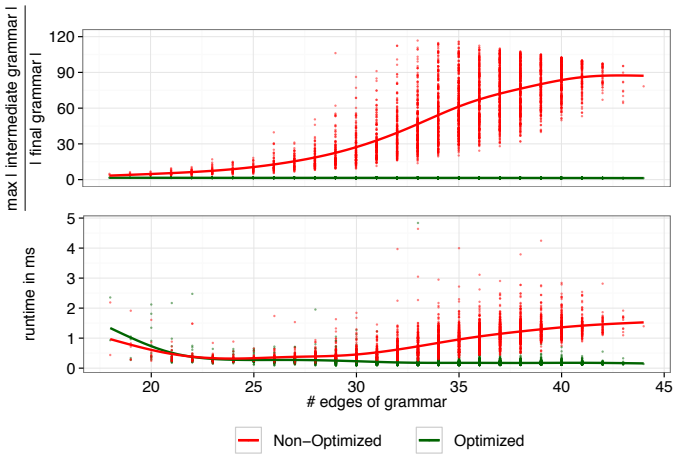


Fig. 3. Effect of our optimization on grammar size and runtime

If we deactivate the optimization, the blow-up seems to grow rather in the size of the original XML tree. For our sample data, we reach a blow-up of more than 110. This can also be observed for the runtime, as the runtime of the non-optimized version scales much worse than the runtime of the optimized version. Only for very small grammars, the runtime overhead caused by the optimization is greater than the yielded benefit, resulting in runtimes including the optimization that are slightly worse than the runtimes without the optimization.

C. Dynamic Compression

In this section, we investigate the use of GrammarRePair as a tool generating a mutable compressed tree data structure. For this purpose, we perform updates on the grammar-compressed tree and then execute GrammarRePair on the resulting grammar. We compare this grammar with the one obtained by first performing the update, then decompressing the grammar, and then compressing the obtained tree.

We consider three update operations: (1) to *rename* the label of a given node, (2) to *insert* a tree before a given node (i.e., as previous sibling of that node), and (3) to *delete* the subtree rooted at a given node.

Since we work on binary tree representations of the original unranked XML trees, we can execute an insert on a null pointer, thus executing an "insert after" the last element (or an insert into an empty child sequence).

More formally, if t and s are binary trees representing XML structures, u is a node of t, and v is the right-most leaf of s (which is necessarily a null pointer), then $\mathit{insert}(t, u, s)$ is defined as $t[u/s]$ if u is a null pointer node, and otherwise as $t[u/s']$ where $s' = s[v/t_u]$ and t_u is the subtree of t rooted at u.

Compression Ratio Comparison. Here, we consider sequences of random insert and delete operations (10% deletes and 90% inserts). The sequences are obtained by starting from a given document, and then applying the *inverse* of the operations until a seed document is derived. In this way, each update sequence starts with a seed document and ends up with an original document from our datasets. This is a well-know technique for approximating realistic update workloads.

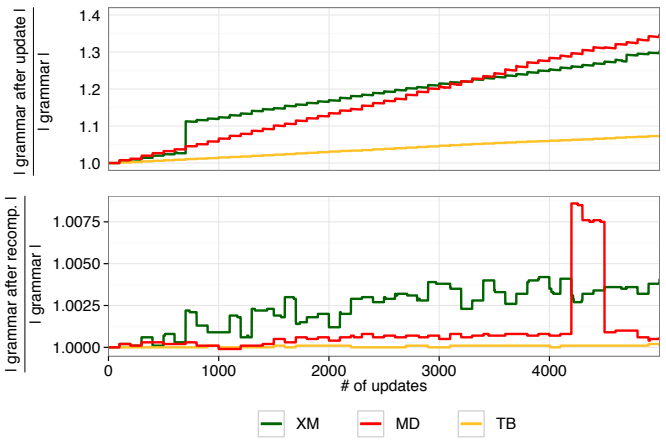


Fig. 4. Update without any recompression (top) and under GrammarRePair (bottom)

As was observed previously, repeated updates to a grammar compressed tree causes rapid degradation of the compression ratio [6]. The reason is that the path isolation process causes decompression of the grammar. We show this effect on a subset of six files: three with very strong compression (NCBI (NC), EXI-Telecomp (ET), and EXI-Weblog (EW)) and three with moderate compression (XMark (XM), Treebank (TB), and Medline (MD)). The results are shown in the top plots of Figures 4 and 5. In a second measurement, we perform the recompression from scratch, i.e., decompression to XML and the compression of the XML to remove the overhead caused by the update after every 100 udpates. We compute the overhead as (size of grammar after update) divided by (size of grammar after recompression from scratch). For the files with moderate compression, the overhead caused by updates is up to 0.4, while for the files with very strong compression, there is a blow-up of up to 400. The latter is because exponentially compressed lists are broken down. The "spikes" in Figure 5 are caused by the decrease of the grammar size after each recompression from scratch.

In contrast, we now run GrammarRePair on the grammar obtained after every 100 updates. Again we compute the overhead as (size of grammar after recompression by GrammarRePair) divided by (size of grammar after recompression from scratch). The results are shown in the bottom plots of Figures 4 and 5. For the three moderate files, we obtain an overhead of less than 0.008. On the extreme files, we obtain a maximal overhead of around 5; recall that the latter grammars are still minuscule compared to the original. Thus, the result shows that GrammarRePair can perform incremental updates and obtains results that are comparable to update-decompress-compress.

Runtime Comparison. In this experiment, we randomly rename 300 nodes of the document to fresh labels (not used in the document). The resulting documents are recompressed using either decompression followed by compression of TreeRePair (gray line), decompression followed by compression of

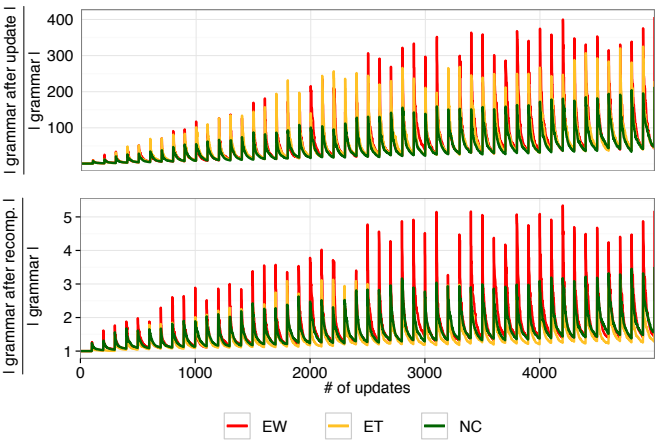


Fig. 5. Update without any recompression (top) and under GrammarRePair (bottom) for exponentially compressing files

GrammarRePair (green boxes) or the recompression of GrammarRePair (red box). Figure 6 shows the results. The number of edges within each XML file and the runtime of 1 are shown below the name. Note that we consider the minimum decompression runtime of TreeRePair and GrammarRePair as decompression time. Although the sequence of decompression and compression of TreeRePair performs better for the smallest file in our tests, it is outperformed by the recompression of GrammarRePair for the larger files. The recompression of GrammarRePair outperforms the sequence of decompression and compression of GrammarRePair for all files. For the largest files, already the compression time of TreeRePair is worse than the recompression time of GrammarRePair. When both applied to trees, TreeRePair only takes $0.1-0.4$ times the time needed by GrammarRePair. This it not astonishing: GrammarRePair is a prototype implemented in Java operating on complicated data structures (as it takes grammars as input), while TreeRePair is a C implementation and optimized for tree input.

In terms of space (not shown), we observe that in the worst case, GrammarRePair uses 23% of the space needed by update-decompress-compress (udc). On average, the space needed by GrammarRePair is much less: it uses only 6% of udc. Thus, GrammarRePair is not only faster, but is also much more space efficient than udc.

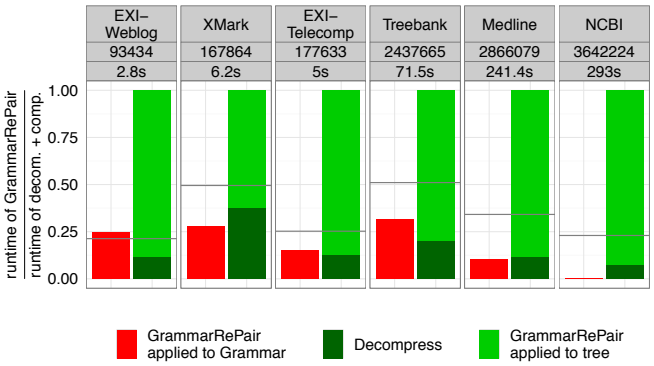


Fig. 6. Runtimes of GrammarRePair versus update-decompress-compress

VI. Conclusions

Straight-line context-free (SLCF) tree grammars offer a compact in-memory representation of XML document trees. Many operations can be supported efficiently over the representation, such as traversals [2, 4], XPath queries[4, 17], equivalence [2], and unification and matching [18]. One major drawback of SLCF tree grammars has been the lack of efficient updates. One way to tackle this problem is to recompress the grammar after an update. In this paper, we present an algorithm that recompresses a given SLCF tree grammar, using the RePair compression scheme. It works surprisingly well in practice: The resulting grammars are as small as compressing from scratch from an uncompressed document. A possible application are compressed DOM representations *with* updates, as DOM is a memory-hungry ingredient of any web browser.

References

[1] P. Buneman, M. Grohe, and C. Koch, "Path Queries on Compressed XML," in *VLDB*, 2003, pp. 141–152.

[2] G. Busatto, M. Lohrey, and S. Maneth, "Efficient memory representation of XML document trees," *Inf. Syst.*, vol. 33, no. 4-5, pp. 456–474, 2008.

[3] M. Lohrey, S. Maneth, and R. Mennicke, "XML tree structure compression using RePair," *Inf. Syst.*, vol. 38, pp. 1150–1167, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.is.2013.06.006

[4] S. Maneth and T. Sebastian, "Fast and Tiny Structural Self-Indexes for XML," *CoRR*, vol. abs/1012.5696, 2010.

[5] N. J. Larsson and A. Moffat, "Offline Dictionary-Based Compression," in *DCC*, 1999, pp. 296–305.

[6] D. K. Fisher and S. Maneth, "Structural selectivity estimation for XML documents," in *ICDE*, 2007, pp. 626–635.

[7] A. Bätz, S. Böttcher, and R. Hartel, "Updates on Grammar-Compressed XML Data," in *BNCOD*, 2011, pp. 154–166.

[8] S. Böttcher, R. Hartel, and T. Jacobs, "Fast Multi-update Operations on Compressed XML Data," in *BNCOD*, 2013, pp. 149–164.

[9] A. Jez, "Faster Fully Compressed Pattern Matching by Recompression," in *ICALP*, 2012, pp. 533–544.

[10] A. Jez and M. Lohrey, "Approximation of smallest linear tree grammar," in *STACS*, 2014, pp. 445–457.

[11] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2554–2576, 2005.

[12] J. I. Munro and V. Raman, "Succinct Representation of Balanced Parentheses and Static Trees," *SIAM J. Comput.*, vol. 31, no. 3, pp. 762–776, 2001.

[13] O. Delpratt, R. Raman, and N. Rahman, "Engineering succinct DOM," in *EDBT*, 2008, pp. 49–60.

[14] S. Joannou and R. Raman, "Dynamizing Succinct Tree Representations," in *SEA*, 2012, pp. 224–235.

[15] G. Navarro and K. Sadakane, "Fully Functional Static and Dynamic Succinct Trees," *ACM Transactions on Algorithms*, vol. 10, no. 3, p. 16, 2014.

[16] Sherif Sakr. XMLCompBench: Benchmark of XML Compression Tools. http://xmlcompbench.sourceforge.net (last accessed: 25.09.2014).

[17] M. Lohrey and S. Maneth, "The complexity of tree automata and XPath on grammar-compressed trees," *Theor. Comput. Sci.*, vol. 363, pp. 196–210, 2006.

[18] A. Gascón, G. Godoy, and M. Schmidt-Schauß, "Unification and matching on compressed terms," *ACM Trans. Comput. Log.*, vol. 12, no. 4, p. 26, 2011.