Squint: Lossy Hierarchical Compression on Symbol Streams

Edward Gan & Max Wang

December 11, 2012

Abstract

Grammar-based compression schemes like SEQUITUR exploit repeated structures in streams of symbolic data to achieve good compression ratios. We develop a set of algorithms for improving upon grammar-based compression schemes by allowing some lossiness in the compression. Although precise subsequences might be altered, in return we can simplify the grammars produced to obtain better ratios while still preserving important syntatic patterns at both the micro and macro levels.

Contents

1 Introduction and Goals

Many sources of data exhibit natural hierarchical structure. For instance, English text often can be sensibly grouped into sections, paragraphs, sentences, and so on, all the way down to words and phonemes. Likewise, music consists of repeated motifs, phrases, themes, etc., which make up musical structure. Programming languages are so hierarchical that their syntax can usually be represented in BNF form. It is natural, then, to represent these types of data using Context-Free Grammars (CFGs), which capture these hierarchical structures; this form of *grammar-based compression* can be used to achieve very good compression ratios.

In the parallel world of image compression, however, two different classes of compression schemes exist—*lossless compression* and *lossy compression*. In lossy compression schemes we allow a certain amount of corruption in order to achieve better compression ratios. Some situations may demand lossless encoding, but many others have less stringent requirements. With texture patterns and some photographs for instance, the exact pixel values are far less important than the overall gradients, colors, and structures in the image. In such cases, lossy compression schemes, such as JPEG for images [9], can be valuable and effective.

We believe it is worthwhile to introduce these notions of lossy compression that are prevalent in image/video compression into the world of hierarchical symbol-stream compression. Introducing lossiness to symbol-stream compression opens up the possibility for better compression ratios.

It may seem strange at first to allow a string such as "mind" to be lossily corrupted into, say, "rind". If, however, one is more interested in recurrent structure in the text than in its precise semantic meaning, these corruptions are unimportant. For instance, when looking at text in a foreign language, one is more likely to pick up the texture of the passage, the repeated sounds, common phrases, and breaks between larger-scale sections of text. In this situation, a change of a single letter would probably go unnoticed. Similarly, when skimming through a huge file, say, an execution log, an administrator usually begins by looking for patterns that stand out. Lorem ipsum is a perfect example of a piece of "text" whose value lies in its texture and sentence/paragraph structure, and we will return to it in later sections.

In such situations, lossy hierarchical compression schemes may prove beneficial if they can provide better compression ratios. Moreover, introducing lossiness to the CFGs which represent the input stream may accentuate more important parts of the stream's structure. Compression of a musical passage might, for instance, capture important motives in favor of showing every rhythm and note. Similarly, compression of text in a particular language may highlight syntactical features. We consider this a relevant potential use of lossy hierarchical compression.

In this paper, we develop two algorithms and implement a broader system for lossily compressing text while preserving its hierarchical structures. The goal was to preserve the feel of the text at both micro and macro levels. Since grammar-based compression schemes were designed to capture these structures, we built our system on top of the existing SEQUITUR grammar inference algorithm. Starting with the CFG generated by a lossless grammar inference algorithm, we repeatedly simplify it, throwing out details along the way to emphasize repetitions. Our main contribution lies in the development, implementation, and evaluation of two algorithms for "simplifying" CFG grammars while attempting to preserve their structure.

2 Background

The work in this paper builds upon existing research in lossless grammar based compression, and also draws inspiration from many techniques used in lossy image and video compression.

2.1 Grammar Compression

As described in the introduction, the proposed research shares many of its goals with the SEQUITUR algorithm described in [8]. The authors of [8] develop an effective algorithm based on iteratively rewriting grammars to keep them small and efficient, and produce interesting grammar-based analyses of texts and musical scores.

The same author in [7] explores a variety of ad-hoc extensions to the SEQUITUR algorithm which improve its compression performance on structured data. These ranged from introducing domain-specific constraints to their grammar, adding a few steps of backtracking to their normally greedy grammar formation, to guessing unifications in attempting to infer recursive grammar rules. Though these may invalidate some of their theoretical results on the asymptotic performance of SEQUITUR, in practice they seem to have worked and add nicely to the grammar inference framework.

The intuition that grammar inference can support many detailed policies is made more formal in [4]. The authors classify the properties a grammar needs to function as a good compressor, and moreover give a set of reduction rules for putting a grammar into an appropriate "irreducible" form. Within this context, grammar inference algorithms similar to SEQUITUR can be described as simple applications of their reduction rules to different ways of generating base grammars. Even more examples of the variety of grammar inference schemes possible under this general framework of reducing grammars can be found in [1]

In summary, established work on grammar inference algorithms provides a solid set of tools for experimenting with the kinds of lossy modifications we propose to make to simplify grammar structure, and we will draw upon the kinds of grammar transformations used in SEQUITUR and in Kieffer's reduction rules.

2.2 Lossy Compression

While we were unable to find work in the area of lossy grammar compression, a number of existing techniques for lossy compression more broadly serve as inspiration.

Kieffer briefly identifies two primary methods for lossy hierarchical compression: wavelet-based and fractal-based schemes [3]. In wavelet-based schemes, a signal is compressed by recording the largest coefficients of its wavelet decomposition, in effect trying to represent a signal as closely as possible using wavelet building blocks. The JPEG image format uses a similar technique in order to compress images lossily [9]. Fractal based compression takes this a step further and represents parts of a signal using self-similarity, encoding an image for instance as a fixed point of contraction maps. This is similar in spirit to what is done in MPEG-1, where parts of one frame can be transformed to encode parts of the next frame. Many of these ideas, such as finding largest matching components of objects, make their way into our algorithms.

Work has also been done regarding the lossy compression of other input formats. For text, methods such as replacing words with synonyms [10] or reordering the middle letters of a word (i.e., all letters save for the first and last) [2] have been explored. However, these are very specific tricks to optimize text, and not more broadly applicable to compressing hierarchical streams.

3 Lossifier Algorithms

3.1 Motivation

In order to make our algorithm for lossily compressing streams as general as possible, we decided to approach the problem of lossy compression by starting with a lossless grammar inference algorithm. By building off of a grammar inference algorithm, we hope to take advantage of some of the structures that are visible in a CFG representation of a stream. In this project, we chose SEQUITUR, but any method of inferring a grammar could be used. After inference, we simplified the resulting grammar, introducing

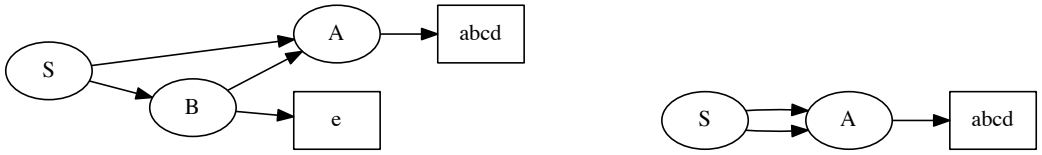

Figure 1: Grammar Simplification Example

loss and obtaining greater compression. Thus our full lossy grammar compression system is a multi-step process:

1. Infer a CFG representation of a stream (we use SEQUITUR).
2. Simplify the CFG, introducing loss.
3. Encode and output the CFG.

Our core contribution comes in step two, via what we refer to as a *lossifier* algorithm—an algorithm that takes an admissible CFG and structurally simplifies it. The output CFG should be smaller than the input CFG but generate a string which preserves the structure in the original string as much as possible. In this paper, we focus on implementing and evaluating this step, and hence omit step 3 altogether.

Building off the ideas in lossy compression discussed earlier, one idea for simplifying data is to look at a part of the data and try to match it as closely as possible with an existing set of primitives. In particular, just as with MPEG and fractal compression, to approximate one part of the grammar we can use *other parts of the grammar*. This is the core principle behind both of our algorithms. Given a rule $A \to \alpha_1, \ldots, \alpha_n$, we will try to replace occurences of A with other similar production rules.

For example, consider the first grammar given in Figure 1. In this graphical representation, we omit the ordering of symbols on the right-hand side of production rules. The picture makes it clear, however, that one easy route for simplifying the grammar is to simply replace the rule B with A. They differ in their expansions by only a single character, 'e', and they form almost identical halves of S.

3.2 Pair-Similarity

To define whether two variables are *similar*, we can look at their expansions. Let $expand(rhs)$ denote the string of terminal symbols we get when we recursively substitute all of the variables inside rhs. Then $V_1 \to rhs_1$ and $V_2 \to rhs_2$ are similar when

$$\frac{levenshtein(expand(rhs_1), expand(rhs_2))}{max(len(expand(rhs_1)), len(expand(rhs_2)))} < \epsilon$$

where *levenshtein* is the Levenstein edit distance between two strings. For convenience we can define.

$$lsim(s1, s2) = \frac{levenshtein(s1, s2)}{max(len(s1), len(s2))}$$

Our first algorithm takes the idea of variable similarity and uses it to iteratively simplify the grammar.

Algorithm 1 Similarity Lossifier Algorithm

```
procedure LOSSIFY-SIM(g)                                   ▷ Simplify grammar g
    while there are similar variables to replace do
        for var_1 → rhs_1, var_2 → rhs_2 ∈ g do
            str_i ← expand(rhs_i)
            if lsim(str_1, str_2) < ε then
                replace the larger var with the smaller throughout g
                break
            end if
        end for
    end while
end procedure
```

3.3 Clustering

The previous algorithm updates the grammar sequentially, since each variable replacement can have an effect on large parts of the grammar, we will have to go through comparing all pairs again after the change. An alternative algorithm would be to try to replace as many similar variables as possible all in one sweep. This would avoid many of the repeat pair-comparisons done in SIMILARITY. In other words, the CLUSTER puts similar variables into equivalence class clusters, and then replaces each variable in a cluster them with the smallest representative all at once each iteration. More detailed pseudocode is given in Algorithm 2 below.

Both of the algorithms depend on the concept of the "smallest / smaller" variable. We can define the size of a variable to be the length of the expansion of its rhs. In this way, if we think of our CFG's as a DAG, then variables higher up on the DAG will always have larger expansions than those lower down, and thus will have a larger size. By always replacing with the smallest variable, we avoid creating cyclic references in our CFG. We will prove this fact in the next section.

3.4 Proof of Grammar Consistency

One risk in replacing variables in a CFG is that one could introduce cyclical expansion in the CFG, such as if $A \to aB$ and $B \to bA$. In this section we will argue that neither of our algorithms do so if the original grammar was well formed.

We can view the CFG as a DAG where for variables V_1, V_2, we add an edge whenever V_1 contains V_2 in its rhs. When sizes are computed on a CFG before we modify it, if $V_1 \hookrightarrow V_2$ is an edge then $size(V_1) > size(V_2)$ since the rhs of V_1 contains V_2. Replacing a variable V_1 with V_2 has the same effect as redirecting all in-edges to V_1 onto V_2.

Suppose for the sake of contradiction that the SIMILARITY algorithm created a cycle at one step by redirecting an edge originally pointing at A to B. This would mean that there was a path going from B to A. However, by comparing the sizes of A and B we know that A must have been bigger than B. Thus there cannot have been a path from B to A, and we have a contradiction.

For the CLUSTER algorithm the analysis is a bit more involved, since to avoid recomputation we only calculate the sizes of the variables once before doing all of the replacements. First note that even as we modify the DAG, all of the edges we create in the DAG will respect the ordering of sizes that we calculated before we did any replacements. This is because each time we redirect an edge from $X \hookrightarrow A$ to $X \hookrightarrow B$ $size(X) > size(A) > size(B)$ and transitivity will preserve the ordering. Suppose then that we created a cycle during CLUSTER by redirecting an edge from $X \hookrightarrow A$ to $X \hookrightarrow B$. This means that there must

Algorithm 2 Cluster Lossifier Algorithm

```
1: procedure CLUSTER-SIM(g)                                   ▷ Simplify grammar g
2:     while there are similar variables to replace do
3:         clusters ← ()
4:         for var_1 → rhs_1 ∈ g do
5:             for c ∈ clusters do
6:                 (var_2 → rhs_2) ← first variable in c
7:                 str_i ← expand(rhs_i)
8:                 if lsim(str_1, str_2) < ϵ then
9:                     add var_1 to c
10:                    break
11:                end if
12:            end for
13:            if var_1 hasn't been added to a cluster then
14:                add var_1 to a new cluster in clusters
15:            end if
16:        end for
17:        save a mapping m from variables to sizes
18:        for Cluster c ∈ clusters do
19:            replace all var ∈ c with the smallest (smallest w.r.t. m).
20:        end for
21:    end while
22: end procedure
```

have been a path from B to X. However, even in its intermediate state we know that the DAG respects the *size* ordering, and $size(X) > size(A) > size(B)$, so there cannot be a path from B to X. This is a contradiction and we are done.

3.5 Runtime Analysis

Given an input of length n, let r be the number of rules generated by SEQUITUR from the input. In the analysis of the SIMILARITY algorithm, let R be the total number of variable replacements we perform. For each replacement, we had to have searched though up to all pairs of variables, and for each pair of variables expanding them and calculating the Levenshtein distance could take time linear in n. Performing the replacement only takes time linear in r. Thus the runtime is $\boxed{O(R * r^2 * n)}$.

From the SEQUITUR paper [8] we know that the number of rules in the grammar generated by SEQUITUR is bounded by $O(n)$. Thus the worst case runtime on the similarity algorithm is $\boxed{O(n^4)}$. This is because we could have up to r replacements. However, in practice we will see that the runtime is much better than $O(n^4)$. There are a variety of possible reasons for this: in our test data, r didn't grow quite linearly in n, R could grow much slower than r, and the use of native code in calculating the Levenshtein distance made its contribution much less noticeable for the input sizes we tested.

The inner loops of the CLUSTER are similar to those of the SIMILARITY algorithm, and an analogous analysis yields a runtime of $\boxed{O(C * r^2 * n)}$. Just as with the similarity algorithm however, a variety of factors make the runtime much better than the $\boxed{O(n^4)}$ worst case scenario. The big gain over SIMILARITY is that C turns out to be nearly constant, since there exist very few huge chains of reductions that must be performed in sequence so we are able to perform many reductions in parallel. Thus a more reasonable

upper bound for the runtime of cluster would be $\boxed{O(n^3)}$

4 Implementation

Our full grammar compression scheme involves the following sequence of operations and components:

1. SEQUITUR, which infers a CFG from the input stream.
2. Lossifier algorithms, which take the SEQUITUR grammar and simplify it, introducing lossiness.
3. Reduction engine, which applies Kieffer and Yang's reduction rules to the lossified grammar.

We implement all these components in the Ruby programming language. Our implementation of SEQUITUR exactly mirrors the data structures and algorithm described by Nevill-Manning and Witten [8]—and, empirically, it has the same asymptotic running time—so we do not discuss it further. However, we will examine in this section some of the techniques we used in implementing SIMILARITY, CLUSTER, and the reduction engine.

4.1 Data Structures and Optimizations

4.1.1 CFG Representation

Our representation of a context-free grammar is very similar to that used by SEQUITUR. All of our rules are represented by a linked list of symbols; however, unlike in SEQUITUR, we omit refcounting and do not point from each instance of a nonterminal symbol to its corresponding rule. Instead, our rules are kept in a hash table, mapping nonterminal symbols to a list representing its RHS.

We chose to implement a list library rather than use Ruby's native arrays in order to achieve better runtime for the splice operation. Both SEQUITUR and our implementation of Kieffer and Yang's reductions require us to splice out a subsequence of terminals from a rule, or to insert a subsequence (e.g., as a result of inlining a singleton rule) into a rule. Moreover, our initial implementation of the `replace` operation—which replaces all occurrences of a nonterminal with another—initially employed rule inlining to prevent the creation of cycles. Our first iteration implementation used native arrays, and we found that linked lists did result in an empirical improvement in runtime.

4.1.2 Lossifier Optimizations

Perhaps the most crucial optimization we implement deals with the recursive expansion of nonterminals into terminal strings—the `expand` operation. An extraordinarily naïve implementation, in which we simply recurse into every rule we find, could have runtime as bad as $O(\prod_{R \in G} |R|)$, the product of the lengths of all rules in the grammar G. We can easily reduce this to $O(\sum_{R \in G} |R|)$ by caching the expansion of each nonterminal when we complete it, and using the cached result for later instances.

We further leverage this caching optimization, however, by preserving the expansion cache across calls to `expand`. This optimization is critical to achieving acceptable runtime for the SIMILARITY and CLUSTER algorithms because of their heavy reliance on expanding. Each iteration of the algorithm requires expanding all nonterminals. Since `replac`ing a nonterminal A with B changes the expansion of all rules containing A, we clear the expand cache whenever replace is called. This is the key reason for CLUSTER's superior runtime relative to SIMILARITY—SIMILARITY needs to expand all nonterms for each individual `replace` operation, whereas CLUSTER clusters `replace` operations, thereby reducing the number of calls and effective overhead of `expand`.

In addition to this optimization, we also use a native C implementation of the Levenshtein function. In early versions of our program, since every pair of variable expansions must be compared, our Ruby implementation of a longest-common-subsequence algorithm was by far the largest bottleneck. Replacing it with a native C implementation of levenshtein distance improved performance dramatically. Though both implementations have the same $O(n^2)$ asymptotic runtime, the constants in the native implementation are so small that calls to levenshtein take nearly negligible time on the data sizes we were able to test.

4.2 Grammar Reductions

Kieffer and Yang's grammar reductions are a collection of rules which, when applied iteratively to a grammar, systematically eliminate internal redundancies, such as duplicate symbol subsequences. They often reduce the size of CFGs by factoring out repeated patterns as new rules and by eliminating unnecessary rules, but they introduce no loss.

Since our lossification techniques make no guarantees about the form of the resultant grammar, unlike, say, SEQUITUR, we produced a naïve implementation of these reduction rules in order to make our lossy grammars as small as possible. As we shall see in the next section, this simple reduction engine has poor runtime and does not scale very well.

5 Results

5.1 Illustrative Examples

It is useful to look at how the algorithms function on a small piece of data with both plenty of hierarchical structure, and also some messiness that most lossless compression schemes would not deal well with. As an example, the string below is similar to what one might come across in a math puzzle book.

```
123,124,123 + 321,323,321 = ???
124,123,123 + 321,3231,321 = ????
--
124,123,123 + 321,3231,321 = ????
123,124,123 + 321,323,321 = ????
```

Disregarding its meaning, it consists of four very similar equations of the form $a + b =?$, and on a smaller scale each summand consists of 3 repetitions of approximately the same string of either 123, or 321,.

When comparing grammars, we chose to define the *size* of a grammar as the sum of the lengths of the rhs's of the production rules. SEQUITUR yields a relatively complicated inferred grammar with 13 rules and a size of 51. Both the SIMILARITY and CLUSTER algorithms with $\epsilon = .4$, yield the following grammar after reduction. It has 5 rules and is almost half the size, at 30.

```
Grammar:
~[*] => ~[BC]--~[BC]?

~[AZ] => ~[A]3~[A]~[A] + ~[C]~[C]3~[C]1 = ???
~[A] => 12
~[BC] => ~[AZ]~[AZ]
~[C] => 32
----------------------
Generated String:
```

```
1231212 + 32323321 = ???1231212 + 32323321 = ???--(cont.)
1231212 + 32323321 = ???1231212 + 32323321 = ????
```

The lossifier algorithm reduced the string to a pair of pairs of the same equations, where each equation consists of the sum of triples of "12" or "32". Smaller discrepencies in the numbers involved were dropped, while the filler commas and linebreaks also ended up being lost. Note that the structure described is easily observable from the grammar generated.

5.2 Similarity vs Clustering

5.2.1 Methodology

For most of our benchmark tests, we focused on two extreme kinds of data: one with large amounts of repetition at all levels, and one with a word and sentence level structure but with little obvious large-scale repetition. One set of test data was constructed by repeating the same letter 'a' n-times but then adding some texture to the data by changing 1/30 of the characters uniformly at random to '*'. These are the *rep-n* test-cases. Another set of test data was constructed from the first n characters of the book of Genesis from the KJV version of the Bible. Call these test-cases *gen-n*.

For our first suite of tests, we compared the effectiveness of the SIMILARITY and CLUSTER algorithms. We might expect SIMILARITY to achieve better quality at the same compression ratio since it updates the state of the grammar between each variable replacement, on the other hand since it must update the grammar it has the potential to run much more slowly than CLUSTER.

In table 1 in the appendix we report the results on running SEQUITUR and CLUSTER, without the grammar reductions, on the rep-n datasets. In table 2 in the appendix we do the same for the gen-n datasets. In these tests we were looking to evaluate the relative fidelity, compression ratios, and runtime of the two algorithms.

In the tables, Time is the wall clock running as measured on a 2.26 Ghz Intel Core 2 Duo Macbook Pro. Ratio is the relative size of the grammar produced compared with SEQUITUR, where the size of the grammar is the sum of the lengths of the rhs's of the rules. *Lev.* is the Levenshtein distance between the output of the lossified grammar and the original input string, it is a kind of measure of the fidelity of our algorithms but doesn't take into account how well patterns were preserved. For the rep-n tests, *Prop.* is the proportion of the final string produced that was '*'. The closer to $1/30 \approx 0.033$, the more accurate the lossifier was in recreating the original texture of the input.

5.2.2 Result Summary

In both datasets, the compression ratios achieved for both algorithms are roughly equal given the same value for ϵ, so it is fair to compare the algorithms for the same values of ϵ. From plotting the data, we see that the runtimes of the algorithms are roughly polynomial in the size of the data, so we can perform a linear regression on the log-log transform of the data to find the exponents α for the runtime $r(n) = n^\alpha$ where n is the length of the input string. In the charts below, all of the linear regressions had R^2 values of over 0.98.

rep-n:

	SIMILARITY		CLUSTER	
ϵ	.4	.2	.4	.2
α	1.859	2.026	1.306	1.305

gen-n:

	SIMILARITY		CLUSTER	
ϵ	.4	.2	.4	.2
α	2.629	2.493	1.814	2.061

In general, CLUSTER has a runtime which is slightly better than quadratic, while SIMILARITY has a runtime which is slightly worse than quadratic. Differant values of ϵ have comparatively smaller effects on the runtime, and both algorithms performed worse on the gen-n dataset. We think this is because there are

more tiny substitutions one can perform on the gen-n dataset, whereas the rep-n dataset allows for much larger scale substitutions which eliminate the need for many small substitutions. As expected, CLUSTER has a significantly better runtime than SIMILARITY, but it is interesting that both algorithms performed much better than our worst case analysis, we believe it is for the same reasons we proposed in our runtime analysis section.

Across all of the tests, the Leveshtein distances were comparable for both algorithms. Though they differed in individual cases, no algorithm did consistently better than the other. In general the Levenshtein distances were high, except in the $\epsilon = .2$ case for the gen-n data where very little compression was possible at all. We believe most of the distance could be explained by the fact that the algorithms could drastically shorten the strings, since it always replaces variables with *smaller* variables as discussed earlier. The proportion of '*'s in the rep-n input was usually the right order of magnitude, with both algorithms making big mistakes occassionally. In most cases, we felt that the two algorithms maintained similarly acceptable fidelity levels in recreating the texture of the original rep-n input file, and we discuss this subjective impression further in the Larger Tests section.

The actual compression ratio was respectable in most tests, with both SIMILARITY and CLUSTER coming in at nearly half the size of the SEQUITUR CFG for the rep-n data, and 70 percent for the gen-n data. This is promising because in [8], the authors show that SEQUITUR on its own achieves better ratios than most standard compression algorithms. A notable exception was the gen-n test with $\epsilon = .2$, where ϵ was so small that barely any substitutions were possible. However, setting higher values for ϵ had diminishing returns for compression for larger file sizes, at the cost of drastically lowering the fidelity of the output. Data with differing amuounts of repetition and noise appear to require different values of ϵ for the optimal fidelity/compression ratio.

In summary, both of the algorithms were able to achieve moderate additional compression ratios on top of SEQUITUR. The fidelity between the original and lossified strings wasn't great as measured by Levenshtein distance, but we think the texture in both the rep-n and gen-n datasets was preserved well. Both fidelity and compression ratio were very sensitive to ϵ depending on the kind of data. The differences between the two algorithms lie almost entirely in their runtime. Empirically the SIMILARITY algorithm tends to consistently incur an overhead of about $O(\sqrt{n})$ on top of the runtime for the CLUSTER algorithm. Because of this we dismiss the SIMILARITY algorithm as unscalable and do not test it further.

5.3 Grammar Reduction Impact

Although the grammars generated by SEQUITUR are irreducible, our lossifiers produce grammars which could potentially be compressed further (losslessly) using Reduction Rules without changing the output string. To test these rules, we looked at the rep-n and gen-n test cases again, this time focusing on the clustering algorithm for $\epsilon = .2$. We report our results in table 3 in the appendix, where we give the ratio on top of the simplified grammar produced by the clustering algorithm, and the time taken to run just the reduction rules.

When plotted on a log-log scale, the runtime for our straightforward implementation of the reduction rules appears to be superpolynomial. While modest additional gains of about 30 percent are possible for highly self-similar data, very few additional gains are possible for more diverse datasets such as gen-n. This appears to be because the only simplifications that are possible on diverse datasets don't lead to the wide-ranging repetitive substitutions which allow reduction rules to be applied.

Because we were unable to get our implementation of reduction rules to scale to larger datasets, we will not consider them further.

5.4 Larger Tests

Though we were ultimately not able to come up with a good way to quantify how much of the structure of a text was preserved, we have included some final output strings obtained after running the CLUSTER algorithm with $\epsilon = .4$ in the Appendices. We believe these tests serve as a good (subjective) summary of the strengths and weaknesses of our algorithm in compressing data while preserving its character.

The first piece of text is from the gen-n dataset again, the first 256000 characters of the book of Genesis. As listed in table 2, the compressed grammar was size 5540 compared with SEQUITUR's size of 8060. Without knowledge of English spelling, the CLUSTER algorithm produces output that is hard to read. However, key words are clearly recognizable, especially important words such as 'God', and the overall division of the book into verses was retained.

The second piece of text is one of the entries to this year's International Obfuscated C code contest, at `http://www.ioccc.org/2012/deckmyn/deckmyn.c` . We believe this is a particularly good example since its structure is foreign, but still discernable. Whitespace is particularly important in standard english but our lossifiers don't treat it any differently from any other symbol. The size of the lossified grammar is 1576 compared with the original size of 1973. The intricate whitespacing was lost, but the clear delineation into sections was kept, key words such as 'define' and 'for' are very visible, and the texture of the obfuscated C is very obvious. For instance, after the distinct '#define' headers, the first section has a bunch of control statements like 'for' and 'if', while the middle sections have very generous spacing and many repeated pointer derence operators such as '***'. Both of these kinds of textures are retained after lossifying.

6 Conclusions

In this paper, we present the design and implementation of original algorithms for performing lossy compression of hierarchical data. To our knowledge, the idea of lossifying context-free grammars as a means of simplifying and compressing hierarchical structure has not been explored in the past. We believe, however, that it is a technique worth further investigation.

We were able to produce a reasonably efficient implementation of two basic grammar lossification algorithms, and in particular, the runtime for our CLUSTER was better than we expected. Our techniques resulted in good compression ratios. The fidelity of the output streams from our compressed grammars, as compared to the original inputs, were lacking—however, our techniques were intended not to maintain the fidelity of outputs, but rather, of hierarchical structure.

We were unfortunately unable to establish numerical metrics for the structural similarities between the lossless and lossy grammars; however, subjectively, we find that lossification maintains interesting invariants, such as the word "God" in a lossy compression of the Bible, or the texture and block structure of an obfuscated C program.

6.1 Future Work

There are many further directions to explore in the area of lossy grammar compression. One immediate direction for future work is to develop additional lossifier algorithms, or to determine techniques for improving our existing ones. One shortcoming of our CLUSTER algorithm is that, by replacing clusters with the shortest rule, we lose both fidelity in the final expanded stream *and* structure in the simplified grammar. Replacing each cluster with the most frequently-appearing rule could improve this situation, at some small cost to the final compression ratio (small because all rules in a cluster are similar in size). Unfortunately, we were unable to devise an efficient algorithm to perform this sort of replacement without accidentally introducing cycles.

Investigating the intuition—or perhaps, providing some methodology—for choosing the parameter ϵ would also be useful. Ideally, we could reliably choose ϵ in such a way that we maximize the ratio of compression to fidelity loss. Investigating the effect that ϵ has for additional types of structured data might shed some light on this problem.

Additional metrics for lossified grammars are also an important direction to consider. Ultimately, our goal is to compress the *hierarchical structure* of data, rather than the data itself, efficiently and with high fidelity. However, we were only able to directly measure how well we preserved structure on contrived data, such as rep-n. On real datasets, we instead made a first-order approximation by comparing the fully-expanded output streams. Some broadly-applicable metrics for comparing hierarchical structures would significantly improve the evaluability of lossy grammar compression techniques.

References

[1] En hui Yang and John C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – part one: Without context models. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 46(3):755–777, 2000.

[2] Yair Kaufman and Shmuel T. Klein. Semi-lossless text compression. *Intl. J. Foundations of Computer Sci*, pages 1167–1178, 2005.

[3] John Kieffer. A tutorial on hierarchical lossless data compression. In Moshe Dror, Pierre Laoecuyer, Ferenc Szidarovszky, and Frederick S. Hillier, editors, *Modeling Uncertainty*, volume 46 of *International Series in Operations Research and Management Science*, pages 711–733. Springer New York, 2005.

[4] John C. Kieffer and En hui Yang. Grammar based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46:2000, 2000.

[5] Eric Lehman and Abhi Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 205–212, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[6] Craig Nevill-Manning, Ian, and I. Witten. Compression and explanation using hierarchical grammars. *Computer Journal*, 40:103–116, 1997.

[7] Craig G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, 1996.

[8] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *J. Artif. Int. Res.*, 7(1):67–82, September 1997.

[9] Gregory K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, pages 30–44, 1991.

[10] Ian H. Witten, Timothy C. Bell, Alistair Moffat, Craig G. Nevill-Manning, Tony C. Smith, and Harold Thimbleby. Semantic and generative models for lossy text compression. *The Computer Journal, Volume 37, Issue*, 2, 1994.

A Data

	Similarity				Cluster			
Dataset	Time	Ratio	Lev.	Prop.	Time	Ratio	Lev.	Prop.
rep-100	0.006	18/34	0.713	0.07	0.002	20/34	0.535	0.04
rep-400	0.028	45/76	0.269	0.01	0.009	41/76	0.743	0.04
rep-1600	0.355	103/194	0.873	0.03	0.042	103/194	0.804	0.02
rep-6400	3.978	258/448	0.706	0.005	0.203	256/448	0.869	0.50
rep-12800	44.63	604/1064	0.791	0.01	1.136	600/1064	0.856	0.01
rep-25600	135.0	900/1482	0.829	0.005	2.906	898/1482	0.725	0.25

(a) $\epsilon = .4$

	Similarity				Cluster			
Dataset	Time	Ratio	Lev.	Prop.	Time	Ratio	Lev.	Prop.
rep-100	0.002	30/34	0.040	0.05	0.002	30/34	0.040	0.05
rep-400	0.019	60/76	0.035	0.02	0.009	60/76	0.035	0.02
rep-1600	0.350	123/194	0.355	0.03	0.055	123/194	0.206	0.01
rep-6400	3.942	280/448	0.387	0.08	0.295	274/448	0.543	0.09
rep-12800	42.21	638/1064	0.491	0.03	1.105	632/1064	0.457	0.01
rep-25600	124.5	922/1482	0.565	0.12	2.597	926/1482	0.437	0.007

(b) $\epsilon = .2$

Table 1: rep-n Tests

	Similarity			Cluster		
Dataset	Time	Ratio	Lev.	Time	Ratio	Lev.
gen-100	0.006	87/89	0.04	0.006	87/89	0.04
gen-400	0.175	224/258	0.12	0.069	219/258	0.19
gen-1600	4.863	539/697	0.31	0.732	529/697	0.33
gen-6400	216.9	1740/2348	0.41	9.127	1694/2348	0.43
gen-12800	1812	3296/4523	0.40	29.17	3181/4523	0.44
gen-25600	12989	5823/8060	0.43	176.7	5540/8060	0.50

(a) $\epsilon = .4$

	Similarity			Cluster		
Dataset	Time	Ratio	Lev.	Time	Ratio	Lev.
gen-100	0.004	89/89	0.00	0.002	89/89	0.00
gen-400	0.085	245/258	0.03	0.053	245/258	0.03
gen-1600	1.292	664/697	0.03	0.874	664/697	0.03
gen-6400	73.83	2173/2348	0.05	14.02	2171/2348	0.05
gen-12800	549.2	4197/4253	0.05	45.64	4186/4253	0.05
gen-25600	3939	7405/8060	0.06	224.9	7374/8060	0.06

(b) $\epsilon = .2$

Table 2: gen-n Tests

Dataset	Time	Ratio
rep-100	0.008	29/30
rep-400	0.028	56/60
rep-1600	0.132	96/123
rep-6400	1.145	153/186
rep-12800	19.43	403/632

(a) rep-n tests

Dataset	Time	Ratio
gen-100	0.016	89/89
gen-400	0.126	245/245
gen-1600	4.833	660/664
gen-6400	308.7	2139/2171

(b) gen-n tests

Table 3: Reducer Tests for $\epsilon = .2$

B Large Examples

deckmyn.c, Cluster, $\epsilon = .4$:

Input:

```
#include<stdio.h>
#define c(C) printf("%c",C)
#define C(c) ((int*)(C[1]+6))[c]
main(int c,char
*C[])          {(C[c]=C[
1]+2   )[0]=   c(52*c(\
'C'+  '4'/4)    );for(c
=0;  c<491;++   c)for(*
*C=  C[1]['c'    +c]  =
0;*   C[0]<8;(    **  C
)++    )C[1][c+   'c']=
*(C[  1]+c+'c')+  C[1][
99+    c]+(C[1    ][**C
+8*c  +99]==32    );  (
*C)[4]=*C[2]==    75  ?
*((C[2]+=3)-2    )==70?
1:0:0;C(0)=C(    1)=c=0
;while(*C[2]?   C[2][1]
?*(C[2]+2)?1    :0:0:0)
{if( *C [2      ]>'w'){
C(1)=0;C[1]    [2]++;*C
[2]=0;}else   C(1)+=*C[
2]==58?2+(    C[2][3]&&
*(C[2]+3)<   'x'):*C[2]
=='s'?(C[   2][1]-=48):
*C[2]>=65  ?3-(*C[2]==\
'm'?1:0)  :1;C(0)=C(1)>
C(0)?C(1  ):C(0);c+=3;*
(C+2)+=3;}printf("  %d\
          %d\n",         56+8*C(    0),80**(C[3]    ++))
         ;*C[2]=0        ;C[2]       -=c;*C[3]       =0;
        while(C[3]       [1,-         1]--){;   for(  **
        C=0     ;*       *C<          80;(**   C)++)  {C
       [2]      -=3      **           C[3];   *C[3]   ++
       =0;      *C[      3]          =**C>=  51||*   *C<
       18       ||*      *C         %8!=2?0  :255   ;c(1
       -1       );c      (*C       [3]);for(       (*C)[
       1]      =0;(      *C)[    1]<3;(*C)[1]    ++)c(*C
       [3      ]|((      *C)[  4]?**C>18&&* *C<42 ?C[1][
       42     +*(*       C+1)    +3***C]:0: **C>=  11&&*
```

```
       *C      <64?       ~C[1 ][   7***C+97 +(*C)[  1]]:
        0)    );c(        *C[3 ]++)    ;for(C (1)=0;   (C(
        2)    =C(1        ))<C (0);)   {(*C) [2]=C   [2][
        1]   -49;         c=(* C[2]<=   63); c=(*   C)[0]
        -4   *(C[         3][0 ]=105-  C[2][ c]   -7*(*(C
        [2]+c)<           'c') -18*(  C[2][c    ]<77)+2*(
         *C)[4            ]-7* (C[2] [c]<'C'  ))-6;for(C(
        3)=0;
       C(3)<                                (*C[2]?*C[2
      ]>'r'                               ?C[    2][1]:(1
     +2*(*C              [2]>   64)    +(2-!C[2    ][3])
    *(58 ==              (*(C         +2))[0])-
    (C[2] [0             ]=='m'   )  ):C(2)?C(0     )-C(
  2):0  );C(3)++)        C(1)+=c(C  [1][4]|(*     C[2]&&
 *C[2  ]<'s'?*C[2]       ==58?C(3)  ==1?**C     >17&&**C
 <51  ?C[ 2]    [1]      ==59?39:  C[2][1     ]==58?9:1:
0:0  :63  >*     C[2     ]?(c<7&&  c>-9?C[1    ][(*C[2]<
45?  'c'  *5     +2*     '%':*C[   2]<  61?570    :571)+
3*c  ]:   0)      :*C    [2]>'o'  ?**C>26&&**C<29    &&!
(*C  )[    2]     ||(    *C)[2]  ==1&&(&*C[0])       [0]
 <34  &&   31     <**    C?C(3)  <2?15+225*C(3     ):0:(
 *C)[  2]  ==     3?C    (3)<2  &&**C>22&&**C     <45&&C
 (3)<      2?     C[     1][7*  *(*C)+151+C      (3)]:0:
  7==(     *C    )[2     ]&&*  *C>26&&**C       <42&&C(3
  )<2?C    [1   ][7      ***C  +135+C(3)]       :0:*C[2]
   <'k'?    (c >-5                    &&      c<5?C(3
    )<2?C   [1][(                     (*C)     [2]<3?
     207:205)+7*                      c+C(3     )]:C[
      2][2]==46              &&       (c==-2||   c==-
            1-                        2*(*C[3]%2  ))?
            96                        :0:0)|((*C)[2
             ]?
             C(          3      )      ?C(3)< 2
             &&                       *C[3]>  7   &&
             c<            1&&c  >-24  ?8:0:*   C
             [3          ]<8&&c  >1   &&c<24    ?   ' '
       :0    :0)         |(C(3) <2     &&(  **  C
      ==66   &&          *C[3   ]>14   ||*  C[  3
      ]>12   &&          58   ===**C||   *
      C[3    ]<          2  &&**C==10   )            ?
       5*  51:                         0)|(7==  (
        *C)[                          2]?*C[3]  <
8?c>13&& c<23&&C(3)<2?C
[1][144+ 7*c+C(3)]:0:C(
3) && c< -14&&c>- 24?C[
1] [7 *c +400+C(3 )]:0:
0) :! C  (3)?**C> 21&&C
[0 ][    0]   <32
?C     [ 1 ]   [(
*   C )    [2   ] +323
   +7     **   *C ]:36
 + 1   < **   C&& '0'>
*     *C ?  C[1][
   C[ 2]  [2]+162
 + 7* **C]:0:0:0));C[1
][ 3] ++;C[2]+=3;}c(0)
;C [3 ]-=2;}*C[3]=0;}}
```

Output:

```
#include<stdio.h>
#define c(C) prinf("%c",C)
#define C(c) (in*)C[]+6))[c]
mainin c,char
*C[])     {C[c]=C
]+  )[0]=  c(52*c(\
C'  '4'/4)   );for(c
=0; c<491;   c)for(*
*  C[]+'c'  c]  =
0;*  0]<8;  **  C
)    C[]+c  'c']=
C[  ]+c+'c'  C[]+99   c]+C[  ][**C
+8*  +99]=3  );
*)[]=*C
]=  7
*C[]+=3)-  )==70?
1):;C()=C(  )=c=0
;whileC
?  2][]+
?C[]+2)?  ):0)
{if(*C [    >'w'){C()=0;C[]+  ]++*C
[2]0;else   C(1)+=C[
2]=58?2+  C
3]&&
*C
+3]  'x'):C

==''?C[  C[]+-=48):
*3]>=6  3-C
]=\
m'?10)  1;C()=C(1)>C(0)?C(  :C();c+=3;*
C[+)+=3;}prinf("  \
      \n",    56+8*C(  0),80**C
  +))
     *C3]=0    C
    -=c*C3]    0;
     whileC
  [1,-    ]+--){  for  **
    =0  ;*  *C      80;(*  C)++)  {C
    3]  -=  **      3]  C[3]  ++    0;      3]      =**>=  51||  *C<    8    ||    *C
            8!=2?0  25  ;c(1    -    );        3]);for    (*)[    ]  0;      <3;(*)[]+
       +)c(*C    [   ]|(      4]**18&&*C<42 ?C[]+    42   (*  C+1)  3***C]0): **  11&&
          *C  64?  ~C
  7***C+97 +(*)[  ]+]:    0)  );c(     ]++  ;forC[ (1)0;  C[    +)  C(    ))<C (0);  {(*)
      3]=C  ][    ]+  -49    c=(* C
<   63); c=(*   )[0]    -4  C[    3][02]105-  ][ c  -7(*C[    ][c)<    'c' -18*  C[]+
    <77)2*    *C)[     ]-7* (C
 [c]<'C'  )-6;forC[    3)0;              C(3]       (C
?C

  ]>'r'     C[  C[]+:(1
  2*      ][  64   +(2-!C
  3])
  (58]=      (*C[    +))[0])-
  C[3] [0    ]='m'  ):C(+)?C(  )-C(  +)0)  ;C(3)++    C()+=c(  C[]+4]|(    3]&&*C2  ]<''?
      C
    =58?C(3)  ]=1?**  >17&&*C <51  C[ ]  ]+    =59?39  C[]+  2]=58?9:1:
```

```
00)  6  >    2   ?(c<7&& c>-9?C
  (*)[
45  'c'    2*  '%':C[  2]  61?570  571)+
3*  ]:  0)  *C  ][’o’  **&&*C<29  &&!
(*     ]  ||  (*)[  ]=&&(&C[0]    [0] <34  &&  3   <**  ?C(3)  2?15+225*C(  ):0:(*C)[
   ]=  3?  C(3)<2  &&**22&&*C  45&&C
C(3)    2?  C[  [7*  (*)+151+    3)]::
  ]=      )[2   &&*C>&&*C    <42&&C(  )<2?  [1  [7  ***C  +135+C(3)    *C3]   <’k’?  c >-
              c<5?C(    2?   []+         (*C  ][3?  207:205)+7        c+C(  ):C
  ][2]]=46          c]=-2||  c]=-    1-        2*(*3]%  ))?    96      0)|((*)[       ]
                   *C            ?C(3)<                    ]>           c<     &&  >-  ?8)
     :            [       <8&&  >1  &&c<       ’ ’       +)  +)    |(C(3) <    &&  **
               ]=66           ]>1   ||*              >1        58  ]=**||
     <       &&*C]=10            5*  51      0)|(7]=          (*)[      2]?C[  <     8?c
     >1&& c<2&&C(3)2?C
C[]+144+ 7*c+C():0:C(
3) && c< -14&&c>- 24C[
1] [7 *c +400+C( ):0:
) :!*C  3)** 2&&C
[0   0]  3
?   [ 1 ]
*   )   2    +323    +7     *C ]:36
 + 1   **  && ’0’>
  *C   C
][      [ 2  ][16
 + 7* **]:):0));C

][ 3] ++C
+=3;}c()
;C [3 ]-=2;}C[]=0;}
```

gen256000, Cluster, $\epsilon = .4$:

Input:

```
1:1 In the beginning God created the heaven and the earth.

1:2 And the earth was without form, and void; and darkness was upon
the face of the deep. And the Spirit of God moved upon the face of the
waters.

1:3 And God said, Let there be light: and there was light.

1:4 And God saw the light, that it was good: and God divided the light
from the darkness.

1:5 And God called the light Day, and the darkness he called Night.
And the evening and the morning were the first day.

1:6 And God said, Let there be a firmament in the midst of the waters,
and let it divide the waters from the waters.

1:7 And God made the firmament, and divided the waters which were
under the firmament from the waters which were above the firmament:
and it was so.

1:8 And God called the firmament Heaven. And the evening and the
morning were the second day.
```

```
(Rest Omitted)
```

Output:

```
1:1 n e beininGod Hae mae ane ar

ine ars itht frm,n void;n darkness aupon
t ofe en.ine Spirt ofGod moend upont ofwate

3e said, htn e s ht

ine w ht, tht waodn God dividhtfrm darknes

5caledht Dayn darkness e caledNht
n e e frinwee e frt

6e said,  amament ine midt ofwate
ndt t dividwatewate

7ine fealn dividwatehichs in
underfeal wates hichs inabenfeal:n t wa

8caledmamenHaen.e e e
frinwee e secon

9e said, undere mae bemae e mae entoone plcen t e dry ln apart wa

0caleddry ln Earn e mae inmae  fwatecalede Sarine tht waod

ine id, Lt e frargras, e ding
seedn e rut dingrut fes inwhseseed i
in tel, e frt wa

n e frrurhtargras, n dingseedfes inn e dingrut , whseseeds in tel, fe
s inine tht waod3e e frinwee e thir

1e said, e behts ine mamene e
todividthayfrm e htn e e e frsignd an
seasond anayn ar: 1:n e e e frhts in
e frmamene e  toinhte fr: n t wa

6ine  tcrathts;e grHatoule thay
nde estoule e hte e e e stars al

7ine t e ine mamene e toinht
e fr1:n toule e e ayn e e htd an
todivide htfrm darknes: God tht waod9n e e n frinwee e fr

0e said, binar aundanolye moilngcaledthhath in frwlthay flyabene frentoopen firmamenofmae

1 n God Haedcratwhalesn enry lcalede

(Rest Omitted)
```