Compression of Partially Ordered Strings

Rajeev Alur, Swarat Chaudhuri, Kousha Etessami, Sudipto Guha, and Mihalis Yannakakis

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Department of Computer and Information Science, University of Pennsylvania
[3] School of Informatics, University of Edinburgh
[4] Department of Computer and Information Science, University of Pennsylvania
[5] Department of Computer Science, Stanford University

Abstract. We introduce the problem of compressing partially ordered strings: given string $\sigma \in \Sigma^*$ and a binary independence relation I over Σ, how can we compactly represent an input if the decompressor is allowed to reconstruct any string that can be obtained from σ by repeatedly swapping adjacent independent symbols? Such partially ordered strings are also known as Mazurkiewicz traces, and naturally model executions of concurrent programs. Compression techniques have been applied with much success to sequential program traces not only to store them compactly but to discover important profiling patterns within them. For compression to achieve similar aims for concurrent program traces we should exploit the extra freedom provided by the independence relation. Many popular string compression schemes are grammar-based schemes that produce a small context-free grammar generating uniquely the given string. We consider three classes of strategies for compression of partially-ordered strings: (i) we adapt grammar-based schemes by *rewriting* the input string σ into an "equivalent" one before applying grammar-based string compression, (ii) we represent the input by a collection of *projections* before applying (i) to each projection, and (iii) we combine (i) and (ii) with *relabeling* of symbols. We present some natural algorithms for each of these strategies, and present some experimental evidence that the extra freedom does enable extra compression. We also prove that a strategy of projecting the string onto each pair of dependent symbols can indeed lead to exponentially more succinct representations compared with only rewriting, and is within a factor of $|\Sigma|^2$ of the optimal strategy for combining projections with rewriting.

1 Introduction

Algorithms for text compression view the input as a linearly ordered sequence of symbols and try to discover repeating patterns so that the input can be represented more compactly. In this paper, we initiate the study of compression of partially ordered strings. Given an independence relation over an alphabet, two strings are said to be equivalent if one can be obtained from the other by repeatedly commuting adjacent independent symbols. An equivalence class of such a type is known as a Mazurkiewicz trace in concurrency theory [Maz87,DM97]. The new compression problem is then to compactly represent an input string if the decompressor is allowed to output *any* string that is equivalent to the original string. For instance, if all the symbols are pair-wise independent of each other, then a string can simply be represented by listing the number of occurences of each occuring symbol of the alphabet in the string. In this case, the original string may be uncompressible, but the extra freedom afforded by independence allows a representation that is logarithmic in the original size.

Many popular algorithms for string compression, such as the Lempel-Ziv algorithms [ZL77,ZL78] and SEQUITUR [NW97], are variant of grammar-based schemes, which work by essentially computing a small context-free grammar that generates the input string uniquely (see [KY00,CLL$^+$02]). Such grammars are deterministic and contain no cycles, and hence can be viewed simply as hierarchical representations of the string. Larus ([Lar99]), using the SEQUITUR scheme, has shown that such compact hierarchical representations of sequential program traces can be used profitably to extract a variety of useful profiling information, such as detection of hotspots and hot subpaths, for analyzing and optimizing a program's dynamic behavior ([Lar99,BL00]).

While executions of sequential programs can be described naturally by strings of events, the behavior of a concurrent system is more appropriately modeled as a partially-ordered sequence of events [Lam78,Pra86,Maz87], reflecting the fact that if events occurring on distinct processes are not causally related their actual order of occurrence may be irrelevant. Message sequence charts (MSCs) offer a visual depiction of message exchanges in a concurrent system, and are used, e.g., for describing high-level requirements in the Unified Modeling Language [BJR97]. MSCs are also best formalized as partially ordered strings. Model checking tools like SPIN [Hol97] generate MSCs as outputs for simulation runs and counterexample traces. Hierarchical representations of MSCs can be used to improved comprehension and visualization of such outputs which are often large. All this suggests that compression of partially ordered strings should be used for concurrent program traces to achieve similar aims as string compression achieves for sequential program executions. In doing so, however, we should exploit the extra freedom provided by the independence relation to find patterns that are not available in a fixed sequential view of a partially ordered trace.

While compression has been studied for decades from both theoretical and practical viewpoints, we are not aware of any research that explicitly addresses compression of partially ordered strings.[1]

Our first class of algorithms involves adaptation of grammar-based schemes directly to partial-order strings. For strings it is NP-hard to find an optimal grammar ([SS82]) but such a grammar is approximable to within a log factor in polynomial time [CLL$^+$02]. We present two algorithms for finding potentially smaller grammar representations by exploiting the independence relation. Our first algorithm is a modification of SEQUITUR ([NW97]) that greedily chooses

[1] Based on the work we have initiated here, S. Savari has begun an information-theoretic study of such structures based on entropy considerations [Sav03a,Sav03b].

the next symbol to be processed from the minimal elements of the remaining partial order by giving preference to the one that would lead to an already encountered pattern. Second is an offline algorithm that repeatedly replaces the most frequently occurring pair of dependent *or independent* symbols by a new nonterminal. As such, it does not strictly speaking produce a string grammar, but rather a limited form of more general graph grammars ([Eng97]). We report on a prototype implementation of these algorithms, and experimental results that indicate improvements in compression.

Our second class of algorithms consists of representing a string by an adequate collection of projections onto subsets of the alphabet, and then compressing each projection by a grammar-based string compression algorithm or by one of the algorithms of the first class. A necessary and sufficient condition for being able to reconstruct the original string up to equivalence is that each pair of dependent symbols must occur in one of the projections. A natural strategy for projection is to project the string onto every pair of dependent symbols. Surprisingly, this strategy can be exponentially more succinct than the optimal representation using just rewriting. In fact, this exponential gap holds even for ordinary strings (that is, when the independence relation is empty). Furthermore, the strategy of projecting onto dependent pairs produces output within a factor of d of that of the optimal algorithm in this class, where d is the number of dependent pairs, and this factor is tight. When the alphabet is partitioned into k sets such that symbols are dependent iff they belong to the same partition, then the natural strategy is to project the input string onto each of the partitions. Compared to compressing the original string, this can be exponentially better in the best case, and it is always within a factor of k compared to the optimal algorithm using just rewriting.

Finally, the third class of algorithms allows collapsing of symbols using relabeling in addition to the projections and rewriting. One strategy in this class is the following. For every symbol a, we project the string onto a and all the symbols dependent on a, then collapse all these dependent symbols to a single symbol b. This leads to $|\Sigma|$ strings, each over a two-letter alphabet, and can be compressed separately. We show how to reconstruct the original string, up to equivalence, from these projections.

2 Grammar-based Compression Up To Equivalence

2.1 Equivalence classes of strings and labeled partial orders

Our model consists of a set Σ of terminals and an irreflexive symmetric *independence relation* $I \subseteq \Sigma \times \Sigma$. Two terminals a,b are said to be independent if $(a, b) \in I$. Intuitively, two strings are equivalent if one can be obtained from the other by a sequence of swaps of adjacent independent symbols. Formally, $\equiv_I$ is the smallest binary equivalence relation on Σ^* satisfying $\sigma ab\tau \equiv_I \sigma ba\tau$, for all $(a, b) \in I$ and for all strings $\sigma, \tau \in \Sigma^*$. We shall represent the equivalence class corresponding to a string σ by $[\sigma]_{\equiv_I}$. Such equivalence classes are called Mazurkiewicz traces in the concurrency literature [Maz87].

Equivalence classes induced by $\equiv_I$ correspond to *labeled partial orders* of a particular form. A labeled partial order respecting I is a structure $P = (V, E, \lambda)$, where V is a finite set of nodes, E is a set of edges over V such that the reflexive-transitive closure E^* is a partial order over V, and $\lambda : V \to \Sigma$ is a labeling of nodes by terminals such that for all $u, v \in V$,

1. if $(u, v) \in E$, then $(\lambda(u), \lambda(v)) \notin I$,
2. if $(\lambda(u), \lambda(v)) \notin I$, then either $(u, v) \in E^*$ or $(v, u) \in E^*$.

A *linearization* σ of the labeled partial order $P = (V, E, \lambda)$ is a string $\sigma_1\sigma_2 \cdots \sigma_{|V|}$ over Σ such that there exists an ordering $v_1v_2 \cdots v_{|V|}$ of the nodes in V satisfying (1) $\sigma_i = \lambda(v_i)$ for $1 \leq i \leq |V|$, and (2) for all $(v_i, v_j) \in E$, $i < j$. We can define a correspondence between equivalence classes of strings and labeled partial orders. Namely, given a string σ and an independence relation I, there is an algorithm to construct the labeled partial order $P_{\sigma,I}$ with $|\sigma|$ vertices whose linearizations are the strings in $[\sigma]_{\equiv_I}$. The details of the algorithm to construct $P_{\sigma,I}$ are standard, and omitted from this abstract.

2.2 Grammar-based compression

In grammar-based compression algorithms for strings, given an input string σ, the algorithm computes a context-free grammar G that generates the singleton language $\{\sigma\}$. The grammar G then serves as a succinct hierarchical representation of σ. From now on, we shall refer to such a grammar as a *grammar for* σ. Over the years, several interesting grammar-based string compression algorithms have been proposed. Of them, the algorithm Sequitur [NW97] has been used for compression as well as to gather profiling information from program executions [Lar99,BL00,GRM03], and is of particular interest to us. Sequitur is an online algorithm that greedily constructs a hierarchy out of an input string. It scans the input from left to right, identifies repeated pairs of adjacent symbols (digrams) in the representation of the input that it has processed so far, and replaces them by nonterminals. A grammar rule maps every nonterminal to the digram it represents.

A good measure of the performance of a grammar-based compression algorithm is the size of the grammar, where the size of a grammar G is defined to be the sum of the lengths of the right-hand sides of all the rules in G. The optimal grammar-based compression algorithm needs to find the smallest grammar for the given input string. Unfortunately, this problem is NP-complete [SS82]. However, some recent research is aimed at finding approximation algorithms for this problem: Lehman et al [LS02] find approximation ratios for some previously proposed grammar-based compression algorithms (e.g., the well known LZ78 has an approximation ratio $O((n/\log\ n)^{2/3})$), and prove the hardness of approximating the smallest grammar beyond a certain constant factor; Charikar et al [CLL$^+$02] present an algorithm with an approximation ratio $O(\log(n/g^*))$, where g^* is the size of the smallest grammar.

2.3 Compression up to equivalence

In this paper, we are interested in generating a small grammar-based representation of a given string *up to the equivalence induced by an independence relation*. We propose three different methodologies for achieving this, and pose three different optimization problems that these methods correspond to.

Finding optimal equivalent strings. In our first approach, we find a string that is equivalent to the input string and can be represented by a small grammar. The output is the grammar for this string. For example, suppose $\Sigma = \{a, b, c\}$ and b and c are independent of each other. Then, the strategy of clustering all the b's (and c's) together between every pair of a's is a good heuristic to increase compressibility. For instance, $abccbacbbc$ will be rewritten to $(ab^2c^2)^2$ to reduce the size of the grammar-based representation. The corresponding optimization problem is as follows. Let $C(\sigma)$ represent the size of the smallest grammar for a given string σ. Then, given a string σ and an independence relation I, the problem is to find $\tau \in [\sigma]_{\equiv_I}$ such that $C(\tau)$ is the minimum of the set $\{C(\sigma') \mid \sigma' \in [\sigma]_{\equiv_I}\}$. From now on, we refer to this optimal value $C(\tau)$ as $C_I(\sigma)$.

Projections and compression. In our second approach, we consider the compression algorithms that project the input string onto a sequence of subsets of Σ such that the original string (up to equivalence) can be recovered from these projections, and compress the projections separately. In the example with $\Sigma = \{a, b, c\}$ with b and c independent, we can represent σ by two projections, one onto $\{a, b\}$ and one onto $\{a, c\}$, and compress the two separately (e.g., $abcccbacbbc$ will be replaced by the pair ($abbabb$, $acccacc$)).

The projection of a string σ on a subalphabet $\Sigma' \subseteq \Sigma$ is obtained by erasing all symbols in σ that are not in Σ', and is represented by $\sigma \uparrow \Sigma'$. Subalphabets $\Sigma_1, \Sigma_2, \ldots, \Sigma_m \subseteq \Sigma$ *cover* an independence relation I, if there is a *reconstruction algorithm* A such that, for all strings σ, given the projections $\sigma \uparrow \Sigma_i$, A outputs some $\sigma' \in [\sigma]_{\equiv_I}$.

In this case, the compression methodology is as follows. We first project the input string σ on a set of covering subalphabets. Then we find grammars for these projections using an approximation algorithm for string compression. The compressed representation of the string (and the equivalence class) is the collection of all these grammars. In order to uncompress, we regenerate the projections from their grammars and use a reconstruction algorithm to generate a string equivalent to σ. Formally, the optimization problem is as follows. Given a σ and an independence relation I, find a cover $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ for I such that $\sum_{i=1}^{m} C_I(\sigma \uparrow \Sigma_i)$ is minimized. Let us denote $\sum_{i=1}^{m} C_I(\sigma \uparrow \Sigma_i)$ for the optimal cover by $C_I^p(\sigma)$.

Relabeling, projections, and compression. In our third approach, we allow relabeling of symbols during projections as long as the original string can be recovered up to equivalence. Going back to our example with independent b's and c's, we can represent a string by a pair, where the first one is a projection onto $\{a, b\}$ and the second one is obtained by renaming b to c. For instance, $abcbccacbcbb$ can be represented by $(abbabbb, ac^5ac^5)$. In this example, it is clear that the original string can be reconstructed up to equivalence, and relabeling can be exploited to minimize grammar sizes.

A *relabeling* γ is a function from Σ to Σ, and we use $\gamma(\sigma)$ to denote the string obtained from σ by replacing each symbol s in σ by the corresponding symbol $\gamma(s)$. A sequence of subalphabets $\Sigma_1, \Sigma_2, \ldots, \Sigma_m \subseteq \Sigma$ and a corresponding sequence of relabelings $\gamma_1, \gamma_2, \ldots \gamma_m$ are said to *cover* an independence relation I if there is a *reconstruction algorithm* A such that, for all strings σ, given renamed projections $\gamma_i(\sigma \uparrow \Sigma_i)$, outputs some $\sigma' \in [\sigma]_{\equiv_I}$. The optimization problem is defined as in the previous case. Given a string σ and an independence relation I, find a set of subalphabets $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ together with relabeling functions $\gamma_1, \gamma_2, \ldots \gamma_m$ such that the two sequences cover I, and $\sum_{i=1}^{m} C_I(\gamma_i(\sigma \uparrow \Sigma_i))$ is minimized. Let us denote the optimal sum by $C_I^{pr}(\sigma)$. Note that, by definition,

$$C_I^{pr}(\sigma) \leq C_I^{p}(\sigma) \leq C_I(\sigma) \leq C(\sigma).$$

3 Compression Algorithms

3.1 Locally greedy algorithm for finding good linearizations.

We first describe an algorithm that takes labeled partial orders as inputs, and outputs grammars for certain "good" linearizations. Given a string σ and independence relation I, we can first construct the partial order $P_{\sigma,I}$. Algorithm of Figure 1 is an online algorithm inspired by Sequitur [NW97] and traverses the input partial order P from top to bottom. At each step, one of the *minimal nodes* (nodes without any incoming edges from unprocessed nodes) is chosen and removed from P. The choice is made greedily by giving preference to a node that will create a digram that has already appeared. Its label a is appended to a list L representing the part of the input already seen. Following Sequitur, we enforce digram uniqueness on L; that is, if a digram xy occurs at two separate locations on L, they are to be replaced by a nonterminal. If this digram has not been seen in the input processed so far, we add a rule $A \to xy$, for some new nonterminal A, to the grammar.

In our implementation of this algorithm, we maintain a map from digrams to positions in L. This map is maintained as a hashtable, so that we are able to match rules in constant time. Changes to the list L – required when a digram is replaced by a nonterminal – are implemented through low-level pointer operations. At each step we contract one edge of the partial order; we terminate when there are no edges left to explore. Since the edge relation is the covering relation of the partial order, there are at most a linear number of edges. If n is the length of the input string, and k is the width of the partial order $P_{\sigma,I}$ (that is, the maximum number of pair-wise unordered symbols), then the algorithm runs in time $O(k \cdot n)$.

```
input  : Labeled partial order $P = (V, E, \lambda)$.
output : Grammar $G$ for some linearization of $P$.
begin
    $G := \emptyset$.
    List of symbols $L := [\lambda(w)]$ for some minimal element $w$ of $P$. Remove $w$
    from $P$.
    Hashtable of digrams $D := \emptyset$.
    repeat
        $Min$ := Set of minimal elements of $P$.
        $p$:= last element appended to $L$.
        if there is $v \in Min$ and digram $A \rightarrow u\lambda(v)$ in $D$ then
            Remove $v$ from $P$. Append $q = \lambda(v)$ to $L$
            Replace the pair $pq$ at the end of $L$ by nonterminal $A$
            If the rule $A \rightarrow pq$ is not already in $G$, then add it. In this case
            there is a previous unreplaced occurrence of $pq$ pointed to by $\delta$ in
            the hashtable. Replace that as well.
            Update D with digrams generated by these changes. If the digram
            uniqueness property is found to be violated, repeatedly replace the
            violating digrams by nonterminals till there is no repetition.
        else
            Choose some arbitrary $v \in Min$. Remove $v$ from $P$.
            Add a digram $A \rightarrow p\lambda(v)$ to $D$ for some new nonterminal $A$. Make
            it "point" to the current last position in $L$.
            Append $\lambda(v)$ to $L$.
        end
    until $Min = \emptyset$.
    $G := G \cup \{S \rightarrow L\}$, where $S$ is a new starting nonterminal.
    Output $G$.
end
```

Algorithm 1: Top-to-bottom

Consider the labeled partial order P corresponding to the string *cabcbac* with a and b independent. Let us follow a run of this algorithm on P. The stages of the algorithm are described in the table in Figure 1. The key step is step 5, where a is preferred over b as it causes a repeating digram.

3.2 Replace most frequent pair.

Our next algorithm is a greedy offline algorithm that chooses the most frequently occurring pair of dependent or independent symbols, and replaces this digram by a nonterminal. Consider a labeled partial order $P = (V, E, \lambda)$. The *frequency* of a pair of dependent symbols (p, q) is the maximum number of edges of the form (u, v) with $\lambda(u) = p$ and $\lambda(v) = q$ such that no two edges share an end-point (note that sharing of end-points can happen when $p = q$); while the frequency of a pair of independent symbols (p, q) is the maximum number of pair-wise disjoint

Step	List L	Comments
1	c	Only one choice.
2	ca	Symbol a chosen arbitrarily.
3	cab	No other choice.
4	$cabc$	No other choice.
5	$cabca$	Choice made to repeat digram ca. Rule $A \to ca$ added.
6	$AbAb$	Symbol b appended. Digram Ab repeated. Add rule $B \to Ab$.
7	BBc	End of partial order reached. Add rule $S \to BBc$.

Fig. 1. Sample run of the Top-to-bottom Algorithm

sets of nodes of the form $\{u, v\}$ such that $\lambda(u) = p$, $\lambda(v) = q$, and neither uE^*v nor vE^*u. The *contraction* of $(u, v) \in E$ by a node w is the following operation on P: remove u, v from V; add w to V; replace $(s, t) \in E$, where $t \in \{u, v\}$ and $s \neq u$, by (s, w); replace $(s, t) \in E$, where $s \in \{u, v\}$ and $t \neq v$, by (w, t); and remove (u, v). For a pair (u, v) of unrelated nodes, the contraction by a node w is defined similarly: remove u, v from V, add w to V; replace $(s, t) \in E$ with $t \in \{u, v\}$ by (s, w); and replace $(s, t) \in E$ with $s \in \{u, v\}$ by (w, t). Finally, we will modify our definition of the labeling function λ a bit so that a labeled partial order can also have nodes labeled with arbitrary nonterminals. The definitions of frequency and contraction apply to such nodes also. If such a new node w is labeled with a new nonterminal A, then A is declared to be dependent on all the symbols that are dependent on p as well as the symbols dependent on q.

At each step of this algorithm, we identify a pair of symbols (p, q) with the maximum frequency. Then we add a rule $A \to pq$, for some new nonterminal A, and contract a disjoint collection of node pairs labeled (p, q) by a node labeled A. Computing the frequency of dependent pairs is straightforward, we simply need to scan all the edges and maintain a count for every pair of symbols. Computing the frequency of independent symbols requires more care, we need to make sure that if a node labeled p is unrelated to two nodes labeled q, then only one pair gets counted to the frequency of (p, q). In this case, matching the p-labeled node with the first possible q-labeled node that is a potential match, is a safe strategy to maximize the count of disjoint pairs. Note that the resulting grammar is not, strictly speaking, a string grammar because we are also allowed to introduce new nonterminals for pairs of independent symbols. Rather, it can be viewed as a limited form of more general graph grammars ([Eng97]), and hence as a generalization of the grammar-based string compression approach to a graph grammar-based approach for compression of partial orders.

Consider again the labeled partial order P corresponding to the string $cabcbac$ with a and b independent. At the first step, we have to choose a set of disjoint edges labeled by the same symbol-pairs. We arbitrarily choose the symbol-pair (a, c) (we could also have chosen (b, c), (c, a), (a, b) or (c, b), all have frequency 2), add the rule $A \to ac$, and contract. The partial order now becomes the one corresponding to the string $cbAbA$. At the next step, we contract the two edges

```
input   : Projections $\sigma_i$, $1 \leq i \leq m$, with $\sigma_i = \sigma \uparrow \Sigma_i$. The following condition
          is satisfied: for all $(a, b) \notin I$, there is an $i$ such that $a, b \in \Sigma_i$.
output  : A string $\sigma'$ satisfying $\sigma' \equiv_I \sigma$.
begin
    $p_i \longleftarrow 1$ for each $1 \leq i \leq m$
    $Proj_a \longleftarrow \{i : a \in \Sigma_i\}$ for each $a \in \Sigma$
    $j \longleftarrow 0$
    repeat
        Select $a \in \Sigma$ such that for all $i \in Proj_a$, we have $p_i \leq |\sigma_i|$ and $\sigma_i(p_i) = a$
        $p_i := p_i + 1$ for all $i \in Proj_a$
        $\sigma'(j) := a$; $j := j + 1$
    until no such $a$ can be selected.
end
```

Algorithm 2: A reconstruction algorithm

labeled (b, A) and add a rule $B \rightarrow bA$. The partial order now becomes a chain cBB. There is no way to contract further.

3.3 Algorithms using projections

The first step in the algorithms that employ projection is to compute a cover for the given independence relation. The next theorem identifies a key property of the cover. (We have been informed that [CP85] contains this result. We provide a proof here for completeness and because our proof provides an efficient reconstruction algorthm which we use.)

Theorem 1. *([CP85]) Subalphabets $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ cover an independence relation I iff for all $(a, b) \notin I$, there is an i such that $a, b \in \Sigma_i$.*

Proof: ($\Rightarrow$) Suppose there is a pair of symbols $(a, b) \notin I$ such that there is no i with $a, b \in \Sigma_i$. Then, the projections of the non-equivalent strings ab and ba will be identical, and hence, reconstruction is impossible.

($\Leftarrow$) For this direction, we give a reconstruction algorithm for any set of subalphabets satisfying the above condition. In Algorithm 2, $\sigma(i)$ represents the i-th symbol of σ. The algorithm keeps a *current pointer* $1 \leq p_i \leq |\sigma_i|$ for each projection $1 \leq i \leq m$. For each projection we advance this pointer from the beginning to the end. The correctness proof is omitted due to lack of space. ■

The reconstruction algorithm, with appropriate book-keeping, can be made to run in time linear in the size of its input (that is, the sum of the sizes of the projections). In the context of this theorem, a reasonable strategy is to project on a set of maximal cliques covering all the edges in the graph for the complement D of the independence relation. We present two special cases of this methodology.

- The algorithm **edge-cover** projects on the set of subalphabets $\{a, b\}$ for every pair (a, b) in the complement D of the independence relation I. This algorithm can be optimized by considering only the pairs (a, b) such that the dependency is realized within the input string σ, that is, the partial order $P_{\sigma,I}$ contains an edge whose endpoints are labeled with a and b.

- An interesting special case is when the independence relation is a k-partite graph: the alphabet Σ is partitioned into sets $\Sigma_1, \ldots \Sigma_k$ such that two symbols are independent iff they belong to separate partitions. In this case, this partition makes a natural choice for the clique cover.

3.4 Algorithms with relabeling and projection

If the subalphabets Σ_j and relabelings γ_j cover an independence relation I, then for $(a, b) \notin I$, there must be an index j such that $a, b \in \Sigma_j$ and $\gamma_j(a) \neq \gamma_j(b)$. That is, a necessary condition for reconstruction is that every pair of dependent symbols must belong to a projection whose relabeling does not collapse them.

Now, we present an alternative to the covering the dependency graph using cliques. Given an independence relation I, for every symbol $a \in \Sigma$, let $\Sigma_a = \{b \in \Sigma \mid (a, b) \notin I\}$ be the set of symbols dependent on a. Let # be a special symbol that is dependent on every symbol, and let γ_a be the relabeling that maps a to a and renames all other symbols to #. The strategy **star-cover** is to project the input string onto Σ_a, and apply the relabeling γ_a before applying the standard string compression. Note that, like the edge-cover algorithm of the previous section, this strategy also leads to a collection of 2-symbol strings, but now, we are guaranteed that we have only $|\Sigma|$ projections, one per symbol in Σ.

Theorem 2. *The subalphabets Σ_a and relabelings γ_a, for each $a \in \Sigma$, cover the independence relation I.*

Proof: The reconstruction algorithm is similar to Algorithm 2. Let $\sigma_a = \gamma_a(\sigma \uparrow \Sigma_a)$. As before, we maintain a pointer p_a for each projection σ_a. At every step, we try to select a symbol $a \in \Sigma$ such that $\sigma_a(p_a) = a$ and for each $b \in \Sigma_a$ with $b \neq a$, $\sigma_b(p_b) = \#$. If such a symbol a is found, the algorithm outputs a, and increments all the pointers p_b for $b \in \Sigma_a$. If there are two such symbols, then they must be independent, and the choice does not matter. ■

3.5 Experiments

In this section, we discuss preliminary experimental results for the top-to-bottom and replace-most-frequent compression algorithms presented earlier. We experimented with two distributed programs shipped as demos with the popular SPIN verification toolkit [Hol97]. One of them (`mobile1`) is a model of a cellphone hand-off strategy, the other (`pftp`) is a flow control protocol. These models consist of a number of processes communicating through message channels. Now consider the natural alphabet of *send* and *receive* events. There is a natural dependence relation on this alphabet: any send is dependent on the corresponding receive. Also, the local clock for every process defines a dependence between

MSC size	Sequitur (random linearization)	Top-to-bottom	Replace-most-frequent
20000	13800	5612	4203
40000	24945	9679	7123
60000	35490	13441	12226
80000	45617	16641	22157
100000	55228	19759	-

Fig. 2. Grammar representations constructed by different algorithms: `mobile1`

MSC size	Sequitur (random linearization)	Top-to-bottom	Replace-most-frequent
20000	7048	4474	3457
40000	12470	7571	5128
60000	17433	10700	12461
80000	22026	13453	15233
100000	27081	15456	-

Fig. 3. Grammar representations constructed by different algorithms: `pftp`

any two sends or receives that it participates in. Such an independence relation induces a special subclass of labeled partial orders called *message sequence charts*. We made SPIN perform random simulations of `pftp` and `mobile1` and produce message sequence charts (MSCs) of different lengths. These MSCs were fed as inputs to implementations of algorithms Top-to-bottom and Replace-most-frequent. We also fed random linearizations of these charts to the string compression algorithm Sequitur. A performance comparison is described in Figures 2 and 3. The tables compare average sizes of grammar representations of MSCs of given lengths. The quadratic-time algorithm Replace-most-frequent did not terminate within a reasonable time for the longest input.

The above results suggest there is a practical advantage in choosing a linearization judiciously (as opposed to randomly). We experimented with this separation more, by studying the performance of the Sequitur algorithm on different linearizations of an MSC outputted by `mobile1` (Figure 4). These linearizations are chosen with various "degrees" of arbitrariness. More precisely, while generating a linearization, we proceed along the lines of the Top-to-bottom algorithm, but make random choices at *some* of the steps. Of course, we cannot hope to generate the entire spectrum of linearizations of a large MSC this way; however, we do seem to get linearizations nicely covering the space between random and greedily chosen linearizations. Note that it is very possible that linearizations with much smaller grammars exist; it is just not easy to find them.

4 Bounds on performance

In this section, we provide some theoretical bounds for the compression problem and strategies mentioned in the previous section.

MSC size	Size of Sequitur output on different linearizations
20000	5612, 7381, 8909, 11584, 13800
40000	9679, 12303, 18911, 21526, 24945
60000	13441, 20121, 27212, 31443, 35490
80000	16641, 23117, 30235, 39318, 45617
100000	19759, 30257, 38221, 47116, 55228

Fig. 4. Impact of the choice of linearization on Sequitur

We first demonstrate an exponential separation between the optimals C_I and C_I^p. We encode the sequence $\langle 0, 1, 2, \ldots, 2^k - 1\rangle$ in binary as follows. Our alphabet is $\Sigma = \{\#\} \cup (\cup_{i=1,\ldots,k}\{b_i^0, b_i^1\})$. The special symbol # will separate two successive numbers in the sequence. The encoding of a number that has 0 or 1 as its j-th bit will have, respectively, b_j^0 or b_j^1 as its j-th bit. That is, we consider the string

$$\sigma = \# b_1^0 b_2^0 ... b_k^0 \# b_1^0 b_2^0 \ldots b_k^1 \# b_1^0 \ldots b_{k-1}^1 b_k^0 \# \ldots .$$

Our independence relation is $I = \{(b_i^p, b_j^q) : b_i^p \neq b_j^q\}$. In other words, distinct b_i^p-s are independent of each other and are all dependent on #. In any string that is equivalent to σ, the set of symbols between every pair of #'s encodes a distinct number $0 \leq n < 2^k$. This makes such a string incompressible using grammar-based algorithms; intuitively, every interval between successive #'s contributes at least one symbol to the grammar. Formally, we show that the application of the Lempel-Ziv compression algorithm [ZL77] to σ compresses it at most by a factor k. Then we will use a relation between $C_I(\sigma)$ and this compressed form proved by Charikar et al. [CLL$^+$02] to show that $C_I(\sigma)$ is smaller than σ by at most a factor of k. Finally, we show that $C_I^p(\sigma)$ is logarithmic in $|\sigma|$.

Lemma 1. *If $\tau \equiv_I \sigma$, then LZ77-encoding of τ is $\Omega(2^k)$.*

Proof: The LZ77 algorithm describes a string w using a sequence $s_1 s_2 \ldots s_d$ of *widgets*. Each widget s_i is either a symbol of the alphabet of w or of the form $s_i = (j, r)$. Intuitively, the latter means "start at the position j of the string encoded by $s_1 \ldots s_{i-1}$ and read the next r symbols." More precisely, a widget (j, r) represents the substring $w(j)w(j+1)\ldots w(j+r-1)$, assuming the length of the string represented by $s_1 \ldots s_{i-1}$ is at least j.

We show there is no way to encode any consistent ordering of σ with fewer than $c2^k$ widgets, for some c. Assume $S = s_1 \ldots s_d$ is the LZ77 form for some ordering τ of σ. Then no widget of the form $s_i = (j, r)$ in S can encode a substring containing two or more occurrences of symbol #. This is because the set of b's that occurs between each pair of #'s in τ is unique, and thus there is no part of $s_1 \ldots s_{i-1}$ that one can refer to obtain the same set, irrespective of how b's are ordered. Consequently, we can have at most two # in the string denoted by (j, r), and thus $r < 2k$. Then the claim holds for $c = 1/2$. ■

The result of [CLL$^+$02] shows that if λ is the length of the LZ77-encoding of a string σ, then $\lambda \leq C(\sigma)\log|\sigma|$. It follows that $C_I(\sigma) = \Omega(2^k)$. Suppose the edge-cover algorithm will project σ onto subalphabets $\Sigma_{i,d} = \{b_i^d, \#\}$, where $i \in \{1, \ldots, k\}$ and $d \in \{0,1\}$. There are $2k$ such subalphabets. It can be shown that each of these projections has a periodic nature and, as a result, a grammar of size $O(k)$. For instance, the projection on b_k^0 and $\#$ is $(\# b_k^0 \#)^{2^{k-1}}$. This shows that $C_I^p(\sigma) = O(k^2)$. Note that the choice of k is arbitrary in the above. Consequently, we conclude the following theorem:

Theorem 3. *For each n, there is an alphabet Σ, an independence relation I and a string σ such that $|\sigma| \geq n$ and $C_I(\sigma) = \Omega(2^{|\sigma|} C_I^p(\sigma))$.*

It is worth noting the exponential separation holds even when the independence relation is empty, that is, even for compressing ordinary strings. Consider the string in the above proof. Clearly, σ itself cannot be compressed. Now, all symbols are pair-wise dependent, and there are $O(k^2)$ projections. It is easy to verify that projections onto each pair $\{b_i^p, b_j^q\}$ is periodic and has a grammar of size $O(k)$. Thus, $C^p(\sigma) = O(k^3)$.

Now we proceed to give an upper bound for the edge-cover algorithm which, given a string σ, projects it onto each edge (a,b) in the complement D of the independence relation I. Let these projections be called $\pi_1, \pi_2, \ldots, \pi_k$. Let $C_I^e(\sigma)$ be the sum of $C(\pi_i)$.

Theorem 4. *For all strings σ and independence relations I, $C_I^e(\sigma) \leq |\Sigma|^2 C_I^p(\sigma)$.*

Proof: Consider the projection π of the string σ on a pair (a,b) of dependent symbols. In the optimal projection based algorithm, one of the covering subalphabets Σ_j must include the pair (a,b). Let τ be a string that is equivalent to $\sigma \uparrow \Sigma_j$. Consider a grammar G for τ. Note that $\tau \uparrow \{a,b\}$ equals π. We can remove all other terminals from each rule of G to get a grammar for π without increasing the size of G. Therefore, $C(\pi) \leq C(\tau)$. Hence, $C(\pi) \leq C_I(\sigma \uparrow \Sigma_j)$, and $C(\pi) \leq C_I^p(\sigma)$. There are at most $|\Sigma|^2$ edges in D, and the result follows. ■

To compress projections of σ onto single pairs of dependent symbols, we can use any grammar based algorithm, in particular, the algorithm by Charikar et al [CLL$^+$02], thereby approximating $C_I^p(\sigma)$ up to factor $|\Sigma|^2 \log(|\sigma|/g^*)$, where g^* is the size of the optimal grammar. The bound for the edge-cover strategy is tight. Suppose $\Sigma = \{a_1, \ldots a_k\}$ such that all symbols are dependent. Consider the string $\sigma = (a_1 \cdots a_k)^n$. The grammar of σ is of size $k + \log\ n$, while the grammar for each $\sigma \uparrow \{a_i, a_j\}$ is of size $\log\ n$, and thus, $C_I^e(\sigma)$ is $k^2 \log\ n$.

An interesting special case for the clique-cover is when the alphabet is union of disjoint alphabets $\Sigma_1, \ldots \Sigma_k$ and two symbols are dependent iff they belong to the same partition Σ_i. In this case, a natural choice for cover is the partition $\Sigma_1, \ldots \Sigma_k$. Let $C_I^c(\sigma)$ denote the sum of $C(\sigma \uparrow \Sigma_i)$. This strategy can be quite beneficial over compressing the original string. For example, if there are two independent symbols a and b, then a random string σ won't compress well, while the two projections onto individual symbols carry the minimal information, namely, the number of a's and number of b's. As the next theorem shows, projecting onto cliques can be worse than compressing the original string, or even an equivalent linearization, by at most by a factor of the number of cliques.

Theorem 5. *If the alphabet is a disjoint union of k cliques, then for any string σ, $C_I^c(\sigma) \leq kC_I(\sigma)$.*

Proof: The proof is similar to the proof of Theorem 4. Consider any clique Σ_i in D, let $\pi_i = \sigma \uparrow \Sigma_i$, and let τ be any string equivalent to σ. Since all the terminals in Σ_i depend on each other, one can show that the size of the optimal grammar for π_i is bounded by the size of the optimal grammar for τ. ■

Again, this bound is tight. Suppose $\Sigma = \{a_1, \ldots a_k\}$ such that all symbols are independent (that is, there are k singleton cliques). Consider the string $\sigma = (a_1 \cdots a_k)^n$. The grammar of σ is of size $k + \log\ n$, while the grammar for each $\sigma \uparrow \{a_i\}$ is of size $\log\ n$, and thus, $C_I^c(\sigma)$ is $k \log\ n$.

Finally, for the star-cover algorithm that uses both projections and relabeling, we can show that every relabeled projection $\gamma_a(\sigma \uparrow \Sigma_a)$ has a grammar of size at most that of the smallest grammar for any string equivalent to the original.

Theorem 6. *For all strings σ and independence relations I, for each $a \in \Sigma$, $C(\gamma_a(\sigma \uparrow \Sigma_a)) \leq C_I(\sigma)$.*

5 Conclusions

In this paper, we have formulated and initiated the study of the compression problem of partially ordered strings. It is worth noting that even for compression of ordinary strings, the use of projections and relabeling, and the resulting succinctness of the representation, has not been studied earlier. While we have shown that projection can lead to exponential succinctness for a class of strings, it remains to be seen if projections, possibly augmented with relabeling, can be engineered to lead to practical general compression techniques.

There are many directions for future research. The application to profiling of executions of concurrent programs, and for visualization large MSCs generated by tools like SPIN in compact form, both seem promising. A recent paper applies standard string compression techniques to parallel program executions [GRM03], and our techniques can potentially improve their results. Compression of partially ordered strings can be studied from an information theoretic perspective. Based on the work we have initiated here, Savari has begun a study of the graph entropy of such structures and of rewriting strings to normal forms [Sav03a,Sav03b]. We would also like to sharpen the approximability of the optimization measures introduced in this paper. In particular, approximability bounds for the measure C_I, and improving the $|\Sigma|^2$ bound for the measure C_I^p, are open problems. Finally, it would be interesting to study compression of labeled partial orders based on more general classes of graph grammars ([Eng97]) than those implicit in Algorithm 1.

Acknowledgements: Thanks to Serap Savari for discussions and comments.

References

[BJR97] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.

[BL00] T. Ball and J. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, 33(7):57–65, 2000.

[CLL$^+$02] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and A. Shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 792–801, 2002.

[CP85] R. Cori and D. Perrin. Sur la Reconnaissabilite dans les monoides partiellement commutatifs libres. R.A.I.R.O.-Informatique Thorique et Applications, 19:21-32, 1985.

[DM97] V. Diekert and Y. Metivier. Partial commutation and traces. In *Handbook of Formal Languages: Beyond Words*, pages 457–534. Springer, 1997.

[Eng97] J. Engelfriet. Context-free graph grammars. In *Handbook of Formal Languages, vol. 3*, ed. G. Rozenberg and A. Salomaa, Springer-Verlag, 1997.

[GRM03] A. Goel, A. Roychoudhury, and T. Mitra. Compactly representing parallel program executions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, 2003.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[KY00] J. Kieffer and E. Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46:737–754, 2000.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[Lar99] J. Larus. Whole program paths. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation*, pages 259–269, 1999.

[LS02] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 205–212, 2002.

[Maz87] A. Mazurkiewicz. Trace theory. In *Advances in Petri nets: Proceedings of an advanced course*, LNCS 255, pages 279–324. Springer-Verlag, 1987.

[NW97] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[Pra86] V.R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1), 1986.

[Sav03a] S. Savari. Concurrent processes and interchange entropy. In *IEEE Int. Symp. on Information Theory*, 2003. To appear.

[Sav03b] S. Savari. On compressing interchange classes of events in a concurrent system. IEEE Data Compression Conference, 2003.

[SS82] J. Storer and T.G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29:928–951, 1982.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.