Algorithms for Compressed Inputs

Nathan Brunelle*  Gabriel Robins  abhi shelat

Abstract

Although data is compressed for storage, most algorithms require explicit and uncompressed inputs. One notable exception is the class of string algorithms for which extensive efforts have been made to develop search and pattern matching algorithms that operate on various compressed formats.

In this paper, we expand those efforts by considering tasks such as sorting, indexing, topological graph sorts, and bipartiteness checking of graphs in the situation when the input has been compressed using popular techniques such as LZ77, grammar-based compression, and Boldi-Vigna graph compression.

In all cases, we discover algorithms that outperform the trivial approach that decompresses the input and runs a standard algorithm. We aim to develop an algorithmic toolbox for all basic tasks that operates on a variety of compression inputs.

1 Introduction

We propose the adoption of a framework by which algorithms are designed to exploit the regularity of input data, and are measured by their ability to do so. To accomplish this, we encourage the development of algorithms that are designed to operate directly on compressed data, i.e., the study of compression-aware algorithms.

There has already been extensive research on compression-aware string algorithms. One of the most well-explored areas is pattern matching on compressed strings, including both exact pattern matching [2, 17, 20, 24, 26], and approximate pattern matching [1, 4, 8, 16, 17, 23, 25]. Others have studied compression-aware edit-distance algorithms [3, 7, 11, 14, 21]. Computational biologists have also written genomics-specific algorithms to run on compressed human genomic data [22]. There have also been algorithms presented which act directly on JPEG-compressed images [12, 27]. Initial steps have been made in compression-aware algorithms on graph compressions in [13, 15].

In this paper we study a class of algorithmic problems for numeric sequences and graphs that benefit when designed to run on data that has been compressed using a variety of schemes including LZ77 [29], grammar-based compression [18, 28], a grammar-based graph compression put forth by [9], and a method presented by Boldi and Vigna in The WebGraph Framework [5].

*University of Virginia, njb2b@virginia.edu, robins@cs.virginia.edu, shelat@cs.virginia.edu
1.1 Problem Statement and Results

Sorting and Statistics The first three schemes presented in this paper consider the classic problem of sorting and statistics. To begin, we consider sorting algorithms under the following three compression schemes: The first is Lempel-Ziv ’77 (called LZ77 throughout). The second represents an array as a context free grammar (called CFG throughout). The final is Lempel-Ziv ’78 (called LZ78 throughout), this approach is similar to that for CFG, so we defer the reader to the full paper for details [6].

For sorting an LZ77-compressed sequence of numbers, we present a sorting algorithm which operates in time $O(C + |\Sigma| \log |\Sigma| + n)$ where $C$ is the compression size, $n$ is the length of the sequence, and $\Sigma$ is the set of unique numbers in the input list. In most instances $C \ll n$, thus our algorithm in practice achieves linear sorting as compared to the classical algorithm’s $O(n \log n)$ worst-case performance. Additionally, at no cost to its asymptotic time complexity, the output can be expressed in LZ77-compressed form. We also present a way of indexing into the sequence in $O(C)$ time.

For sorting a list compressed by a context-free grammar we present an algorithm which finds the sorted sequence in $O(C \cdot |\Sigma|)$ time. Here, $C$ represents the total number of symbols in all of the grammar’s substitution rules. This result has the advantage of being independent of the size of the uncompressed list. From here, we can produce a grammar for the sorted list which has size $O(|\Sigma| \log n)$, where $n$ is the length of the decompressed list. The classical approach would require $O(n \log n)$ time to decompress and then sort.

The run-time goal for all these sorting and statistics algorithms is to be independent of the uncompressed sequence size $n$. This is achieved in the cases of LZ78 and context free grammar sorting, and thus we believe these algorithms to be optimal. We leave as an open question whether a similar result can be achieved for LZ77.

Algorithms for Compressed Graphs The final two schemes presented consider the problems of topological sort and bipartite assignment on graphs. The first scheme is a graph interpretation of the Re-Pair compression scheme, and the second is the compression scheme presented by Boldi and Vigna as part of the WebGraph Framework (called BV throughout).

For a graph compressed using the Re-Pair algorithm, we perform bipartite checking and topological sort in $O(C)$ time, where $C$ is the size of the compressed graph. We use the fact that a Re-Pair compressed graph itself can very easily be conceptualized as a graph and be operated on directly. The improvement we obtain for both algorithms is over the $O(|V| + |E|)$ time for the classical approach.

We present an algorithm which can perform bipartite checking on a BV compressed graph using disjoint set data structures. This algorithm runs in $O(|V| + s)$ where $|V|$ is the number of vertices in the graph, and $s$ is the total number of sequences given in the compression scheme. This is a benefit over the running time of the classical approach of $O(|V| + |E|)$.

The performance of the web graph compressions is limited by the inability
for the schemes to reduce vertex size. For this reason we believe our results for Re-Pair and BV compression to be near optimal.

In both cases, our graph algorithms make fundamental use of properties implied by the compression scheme to speed up the analysis of the graph. In particular, for bipartiteness, both compression schemes encode redundancies in a manner which allow certain bipartite checks to be skipped. At a high level, in the same way that the uncompressed version of the graph may have statistical redundancies allowing for compression, an algorithm operating on the uncompressed data may perform unnecessary operations or explore unnecessary branches of a search tree. The structure of the compressed data often reveals properties of the input which may be exploited for more efficient algorithms.

**Definitions** In the case of strings, the notion of a suitable compression scheme has been widely studied and well-defined. In the case of graphs, however, we are not aware of such consensus among definitions. Feder and Motwani [13] present a working definition of a graph compression, repeated here:

Let \( G(V, E) \) be a labeled graph with \(|V| = n\) and \(|E| = m\). A compression of \( G \) is defined as a labeled graph \( G^*(V^*, E^*) \) such that:

1. \( n^* = |V^*| \) is polynomial in \( n \).
2. \( m^* = |E^*| \) is significantly smaller than \( m \), i.e. \( m^* = o(m) \).
3. The mapping \( * : G \rightarrow G^* \) is 1−1.

The long-term goal of this project is to design compression schemes at the same time as the algorithms which operate on the compressed data, and unfortunately, the definition above is not suitable because the (weak) restriction that \( n^* \) need only be polynomial in \( n \) allows trivial ”compression-aware” algorithms. For example, in the case of bipartite assignment on Re-Pair compressed graphs, one can imagine a compression scheme which works as follows: (1) assign each node in the original graph a color as per bipartite assignment, (2) compute the Re-Pair compression of the graph, (3) append a special dellineator ♠ and the list of red nodes to the end of the compressed sequence. This compressed representation allows for a trivial algorithm which computes bipartite assignment by simply returning everything after ♠. However, any such scheme adds redundancies to the encoding.

As with the case for string compression, the correct notion for graph compression must relate the ability for an encoder to asymptotically approach the entropy rate for a wide class of graph generator models. In that case, the above trivial solutions would not be considered good compressors because the amount of added redundancy would not allow the compression rate to approach the entropy. Nonetheless, in our survey of the literature on compressed graphs, we are unaware of schemes that identify such a ”class of graphs” and prove that their compression schemes are asymptotically optimal with respect to those graphs. As a compromise, we only study graph compression schemes which appear in the literature, and also satisfy the weak definition of Feder and Motwani.
Organization We cover numerical compression-aware algorithms in sections 2 and 3 and graph algorithms in sections 4 and 5. We list and prove all theorems and lemmas and provide pseudocode in an extended version of this paper [6].

2 Lempel-Ziv '77 Compression

LZ77 scheme An LZ77 compression consists of a sequence of terminals or back pointers. A terminal is a character from the original set of values $\Sigma$ of the array, and a back pointer is of the form $(\text{back}, \text{length})$ where $\text{back}$ is the index (in the uncompressed array) from which to start a subsequence and $\text{length}$ is the number of characters to copy starting from $\text{back}$. A back pointer may have $\text{length}$ larger than the depth of $\text{back}$ (that is $\text{back} - \text{current_index}$, where $\text{current_index}$ is the uncompressed lists' index of first character of this term). In this case the referenced string is repeated to fill in the gap. For example, if we have the compression $a \ b (1, 6)$, this would decompress into: $a \ b \ ab \ ab \ ab$.

Sorting The intuition behind our LZ77 sorting algorithm (LZ77.Sort in the full version) which sorts an LZ77 compressed array consisting of $n$-bit integers\footnote{The alphabet is considered to be the set of $n$-bit integers. In a future work, we aim to consider the more realistic case of a smaller alphabet (e.g., 8-bit strings) in which the numbers are expressed in multi-precision.}, is that for any sequence $s$, where $lz(s)$ is a LZ77-compression of $s$, the unique set of terminals present in $s$ is equal to those in $lz(s)$. Therefore we begin by first copying all literals to a list and sorting this list.

The next step is to determine the frequency of each literal in the decompressed array. The trivial method to accomplish this takes $O(n)$ time and $O(n)$ space. By utilizing a circular buffer our method slightly reduces the space requirement as follows: we count characters while mimicking the action of the decompression. We scan through the compression to find the length of the deepest back pointer (the maximum $\text{back} - \text{current_index}$). We then create a circular buffer of this size (call this variable $\text{size}$). Now we perform a decompression of the string, except whenever we would append a character to the decompression we instead write that character to the next space in the buffer, and iterate a counter for that character. When reading a back pointer we begin copying from that location in the circular buffer. Since the length of the circular buffer is the depth of the deepest back pointer, we are guaranteed the reference is in the circular buffer. The time complexity of this algorithm is $O(C + |\Sigma| \log |\Sigma| + n)$. The circular buffer allows this algorithm be run in space $O(C + \text{size})$. Note that in most practical implementations of LZ77 compression, the variable $\text{size}$ is a fixed constant.

With the multiplicity of each character we can then return a LZ77 compressed sorted string in time $O(|\Sigma|)$. Assume $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$, and that $\sigma_i$ has multiplicity $m_i$ in the string. Then the compressed string becomes: $\sigma_1 (1, m_1 - 1) \sigma_2 (m_1 + 1, m_2 - 1) \ldots \sigma_{|\Sigma|} (n - m_{|\Sigma|}, m_{|\Sigma|})$.

Indexing The LZ77 indexing algorithm, called LZ77.Index in the full version, gives a method for finding the character at index $i$ of an uncompressed array given a LZ77 compressed array. Combined with the sorting results above, this
algorithm can be used to find the $i^{th}$ order statistic.

The algorithm maintains two indices: $j_{comp}$ and $i_{raw}$. The one labeled $j_{comp}$ reads the current location in the compressed list, the one labeled $i_{raw}$ represents the location in the decompressed string if all terms up to $j_{comp}$ were decompressed. The algorithm operates in two stages. In the first stage it scans through the terms in the compressed list searching for the value of $j_{comp}$ for which $j_{raw} \geq i$, i.e. the term in the compression containing index $i$ in the uncompressed data. If this term is a terminal then this value is the solution, otherwise we continue to stage two. This stage updates the value of $i$ based on the back pointer and scans backward through the compression until a terminal is reached. For example, if $i = 11, j_{raw} = 13$ on term $(3, 4)$, then index 11 is at index 1 in the term, thus the terminal at 11 is the same as the terminal at 4. These two stages together yield a final run time of $O(C)$.

**3 Context Free Grammar Compression**

A (context free) grammar is a 4-tuple $(\Sigma, V, S, \Delta)$, where $\Sigma$ is a finite set of values called terminals, $V$ is a set of variables, $S \in V$ is a special start variable, and $\Delta$ is a set of rules (for our purposes only one rule is permitted per variable). A rule is of the form $v \rightarrow s$ where $v \in V$ and $s \in (V \cup \Sigma)^*$ is a sequence of terminals and variables called the definition of $v$. The grammar is read by iteratively substituting variables for their definitions (starting with $S$) until only terminals remain. For example, the string $aabbbabbbb$ can be translated into the grammar:

$$A_0 \rightarrow aA_1A_2A_3, \ A_1 \rightarrow ab, \ A_2 \rightarrow A_1b, \ A_3 \rightarrow A_2b$$

Lempel and Ziv in 1978 presented a secondary compression scheme (LZ78) which acts as a restricted version of CFG compression. The algorithms presented for CFG may undergo simple modifications to operate on LZ78. The details are outlined in the full version of this paper.

**Sorting** Similar to LZ77.Sort for sorting LZ77 compressed arrays, sorting a CFG compressed list exploits the fact that all literals in the uncompressed list occur in the compression. Therefore we begin by first finding and sorting $\Sigma$. Next we turn the grammar into a dependency graph. For this, we say that if the variable $v_0 \in V$ has in its substitution rule $v_1 \in V$, then $v_0$ depends on $v_1$. This graph must be acyclic (since otherwise the grammar would produce more than one string), thus a topological sort exists.

We then consider each literal as a vector of $|\Sigma|$ dimensions such that for the minimal element $\sigma_1 \in \Sigma$, $\sigma_1 \mapsto (1, 0, \ldots, 0)$, and the next smallest element $\sigma_2 \mapsto (0, 1, 0, \ldots, 0)$, and so on. As a notational convenience we say that $\langle \sigma \rangle$ refers to the respective vector for symbol $\sigma$.

The final step is to follow backwards through the topological sort and sum up each symbol’s respective vector upon discovery. In the example given we begin with $a = (1, 0)$ and $b = (0, 1)$. We then calculate the vector $\langle A_1 \rangle = \langle a \rangle + \langle b \rangle = (1, 0) + (0, 1) = (1, 1)$. We can then calculate $A_2 = (1, 2)$. Eventually we calculate the start symbol $A_0 = (4, 6)$, which indicates that in the decompressed string there are 4 $a$’s and 6 $b$’s.
Figure 1: A sample graph (a) and its representation as a Re-Pair compressed graph (b). The dot-circled vertices are dictionary keys and the destination their outgoing edges are the edges which they replace in the adjacency lists.

We are now able to return to the user a context free grammar of size $|\Sigma| \log n$. This is done through standard doubling methods in which at most $\log n$ variables are used to represent correct string for each member of $\Sigma$.

The time complexity of this sorting algorithm is $O(C \cdot |\Sigma|)$, the space complexity is $O(C)$.

4 Re-Pair for Graphs

Much of the existing work in graph compression schemes has been done on web graphs. In these graphs vertices represent web pages, and edges represent hyperlinks from one page to another. Web graphs are therefore directed graphs with unit-weight edges. In [9] the authors introduce an adaptation of the Re-Pair dictionary compression scheme originally presented in [19]. In the latter the authors observe that their compressed graph allows “fast navigation”, we show that this property also parallels fast breadth-first search and depth-first search, and thus faster implementations of algorithms based upon them.

In this scheme (as well as the Boldi-Vigna scheme to be discussed) each vertex is assigned an index $i \in \mathbb{N}$. The graph is then represented in adjacency list form, and is sorted by index. This can be done using a list of lists, where list $i$ contains the destinations of all edges from vertex $i$. Next the method finds the $k$ most common pairs, and introduces a unique symbol to represent each. Afterward each appearance of each pair is replaced by its respective special character. This repeats to satisfy one of two conditions: each pair in the adjacency list appears exactly once, or some pre-defined number of passes is reached.

Once this substitution has been accomplished we represent the compressed data itself as a graph. Each dictionary variable appears as a vertex in the graph with an edge to each vertex it represents, an edge whose destination is a dictionary vertex is assigned zero weight. This reveals the interesting advantage of this compression scheme. Since the compressed graph can itself be represented as a graph many classical algorithms require only minor adjustment to operate on the compressed data. However, this compression scheme only serves to reduce the number of edges in the compressed graph, the number of vertices is larger than those in the original. So if $V_C$ and $E_C$ are respectively the sets of vertices edges in the compressed graph, $|V_C| \geq |V|$ and $|E_C| \ll |E|$. The compression may be seen
as a trade-off from a dense graph with fewer vertices in favor of a much more sparse graph with more vertices. Thus there may be no benefit to operating on the compressed data versus the uncompressed if the algorithm requires high time complexity in terms of number of vertices. In general the running time of graph algorithms is expressed in terms of some function of the number of vertices, $|V|$, and the number of edges, $|E|$. An algorithm run on Re-Pair compression should run faster than its uncompressed counterpart for any algorithm $A$, with running time $f(|V|,|E|)$, where $f(|V|,1) \in O(f(1,|E|))$. Below, we consider two examples when this is beneficial. Additionally, being a virtual node compression scheme, algorithms run on the Re-Pair graph compression scheme could utilize the technique shown by Karande, Chellapilla, and Andersen [15] for fast multiplication of adjacency matrices by a vector.

**Topological Sort**  The method RePair\_Topological (shown in full paper), given a Re-Pair compressed directed acyclic graph, returns a topological sort of its uncompressed graph. The topological sort of the uncompressed graph is embedded in that of the compressed graph. This follows since all vertices in the uncompressed graph are present in the compressed, and for any two vertices $u, v$ where $u$ appears before $v$ topologically in the compression $u$ must also appear before $v$ topologically in the uncompressed counterpart. Thus we may perform a topological sort on the compressed graph, then remove the dictionary terms from the sort. This procedure takes time $O(|V_C|)$. It requires space $O(C)$.

**Bipartite Assignment**  Many of the algorithms performed on unweighted directed graphs such as web graphs can be written with minor modifications of breadth-first search (BFS) or depth-first search. To demonstrate the ability to perform meaningful applications of breadth first search we present an algorithm which determines whether or not a given compressed graph is bipartite (i.e., each vertex is assigned one of two colors such that there is no edge between two nodes of the same color), and if so it appropriately assigns colors to vertices. The classical algorithm for this problem performs a modified BFS which, when a vertex is discovered, assigns that vertex a color opposite of its parent’s. If a vertex is already discovered and is the same color as its parent then the procedure terminates and returns error, as the graph is not bipartite.

We modify this algorithm to work on Re-Pair compressed graphs. The intuition utilized is that a virtual vertex (one representing a dictionary variable) must have the same color as all nodes it represents. Each vertex not representing one of the special characters (an uncompressed graph vertex) is treated the same in the compression-aware version as in the standard algorithm. Each vertex must have color opposite its parent’s. The opposite must be true, however, if the child vertex is a virtual vertex. Since the virtual vertex is a “stand in” for vertices which would otherwise be connected to the parent vertex, this vertex’s color should match that of the parent vertex. This way, once the BFS reaches the virtual vertex, the incident vertices will be defined opposite that of the virtual vertex and thus opposite that of their uncompressed graph parent’s.
5 Boldi and Vigna: WebGraph Framework

Boldi and Vigna in [5] present a compression technique as part of their WebGraph Framework. This scheme first requires that all vertices be indexed in some order. The graph is represented in its adjacency list form, with the vertices in this order. The adjacency list is compressed by representing destination vertices in three ways: copy blocks, intervals, and residuals.

Each element in the adjacency list has a reference element. The copy blocks field gives an encoding of which vertices in the current element’s adjacency list are shared with those in the referenced element’s. The interval’s field gives a list of sequential blocks such that all vertices in the block are in the adjacency list, for example it may state that “all vertices 1-5 and 13-24 are in this element’s adjacency list”. The residuals are the nodes which cannot be expressed in any of the other two ways. For simplicity we consider residuals as intervals of length 1.

Bipartite Assignment The advantage of the BV compression comes from the representation of the copy blocks and the sequences, thus any algorithm which effectively utilizes the compression must utilize these values atomically. While performing bipartite checking we know that at a given vertex the color of the back reference must match that of the current node, as otherwise there would be conflict with the destination nodes they share and imply the graph is not bipartite. We exploit this property in order to skip the copy block nodes. At a given node we mark the back reference as the same color, but indicate it is still unvisited to be processed later.

We use a disjoint-set forest data structure which supports the operations MAKE-SET, UNION, and FIND in time $O(m \cdot \alpha(n))$ where $m$ is the number of all these three operations, $n$ of which are Make-Set, such as the one described in [10]. As a sequence is visited we create a set for each element in that sequence then union all together. If there is another set with nonempty intersection with this sequence, we union both sets together. We maintain four instances of this data structure: two for red vertices and two for blue. For each of these we maintain identical copies, with the maximum vertex as the set representative in one instance and the minimum vertex in the other. We call these forests $max\_red$, $min\_red$, $max\_blue$, and $min\_blue$. By stating red or blue we imply that the same operation be done on both $max\_red$ and $min\_red$ or $max\_blue$ and $min\_blue$.

$BV\_Bipartite$, our algorithm for bipartite checking a BV-compressed graph, operates as follows. We begin with vertex 1 and color it RED. Note that vertex 1 does not have a back pointer in this form of compression. We then add vertex 1 to a list of visited nodes (called $visited$) and red. Next we add each of the sequences in vertex 1’s adjacency list to blue. We also add these vertices to a buffer.

Next we remove an element from the buffer and add that element to $visited$. We then check the color of this element by doing a FIND operation in red and blue. After this we check the color of the back reference vertex by doing a find operation in red and blue. If the color of the back reference does not match that of the current vertex then return error, as the graph is not bipartite. Otherwise if the back reference vertex is uncolored we add it to either red or blue, whichever
the current vertex is in (assume that the current node is blue). We then process each sequence in the vertex’s adjacency list. We traverse parent vertices in both the \texttt{max\_blue} and \texttt{min\_blue} forests to see if any subset has a representative within the sequence. If so then we return \texttt{error}. Otherwise we operate similarly in \texttt{red}, and for any sets not disjoint with the sequence we perform a union of all these sets with any additional vertices from the sequence which must be added. We then add all yet unvisited vertices in each sequence to the buffer. We repeat until the buffer is empty.

In total, this algorithm requires time $O((|V| + s) \cdot \alpha(|V|))$, and space $O(C)$. The uncompressed approach requires $O(|V| + |E|)$ time and space. Since $s \ll |E|$ and for any $V$ of practical size we can consider $\alpha(|V|)$ to be a small constant, this algorithm is much faster than the uncompressed approach, and requires less space.

6 Conclusion
The primary contribution of this paper is its presentation of various sorting algorithms and graph algorithms which operate on compressed data. These algorithms guarantee a time complexity improvement over the classical approach whenever $C \ll n$ for numerical sorting, or in most occasions where $C \ll |V| + |E|$ for graph algorithms. Most importantly we demonstrate that general-purpose algorithms can be performed on compressed data with substantial benefits, and that the benefits often arise from properties exposed by the compression. We broaden the scope of previous results to show wide-spread applicability of a compression-aware approach across many problem domains. This demonstrates the need for developing new theory and techniques for compression-aware algorithms to unlock this potential.

References
10


