Efficient Lossless Compression of Trees and Graphs *

Shenfeng Chen and John H. Reif
Department of Computer Science
Duke University
Durham, NC 27708

Abstract

In this paper, we study the problem of compressing a data structure (e.g. tree, undirected and directed graphs) in an efficient way while keeping a similar structure in the compressed form. To date, there has been no proven optimal algorithm for this problem. We use the idea of building LZW tree in LZW compression to compress a binary tree generated by a stationary ergodic source in an optimal manner. We also extend our tree compression algorithm to compress undirected and directed acyclic graphs.

1 Introduction

Data compression algorithms have been widely used in many areas to meet the demand of storage and transfer of large size data. Most of the data compression algorithms regard the input as a sequence of binary numbers and represent the compressed data also as a binary sequence. However, in many areas such as programming language (e.g. LISP and C)and compiler design, it is more desirable to have a compression algorithm which compresses a data structure which is not a binary sequence and which keeps similar data structure in compressed form as the original data. In this paper, we study the problem of compressing tree and graphs into a similar but smaller form so that many properties of the original data structure are kept in the compressed form. Undirected and directed graph are widely used in representing data structures in programming languages. For example, directed graph is a natural choice for expressing recursive relations. In Holm's system [5], recursively typed languages are represented by directed graphs. Thus the task of checking a given recursive type is reduced to a pattern matching problem on the directed graph. In addition to reducing storage space, a compressed graph also has the benefit of searching for a

*Email addresses: reif@cs.duke.edu and chen@cs.duke.edu. This work was supported by NSF Grant NSF-IRI-91-00681, Rome Labs Contracts F30602-94-C-0037, ARPA/SISTO contracts N00014-91-J-1985, and N00014-92-C-0182 under subcontract KI-92-01-0182.

pattern match more efficiently than in the original graph. Therefore, a compressed graph may be useful to efficiently execute operations on graphs.

Our paper is organized as follows. In section 2, we first define notations and terminology used throughout this paper and the problems with which we are concerned. Section 3 presents an algorithm for compressing binary trees. The algorithm is readily extendible to general trees. We will show that the tree compression algorithm gives an optimal compression for the original tree when size of the tree grows to infinity. In section 4, we will extend our algorithm to compress undirected graph and DAG (directed acyclic graphs). Section 5 gives the conclusion and future work.

2 Background and Terminology

2.1 Tree and Graph Data Structure

An undirected graph $G = (V, E)$ consists of a set of *vertices* V of size n, and a set of *edges* E of size m. Each edge is an unordered pair (v, w) of disjoint vertices v and w. The standard representation of a undirected tree is to use a linked list representing edges connecting to a node. The total number of bits required to represent the graph is therefore $O((n + m) \log n)$. However, this representation is not efficient in the case where the graph is a binary tree. Let T denote a binary tree. We can use the following encoding method to reduce the storage space for T to $O(n)$. Recall depth first search of T has a unique path. We define an alphabet $\mathcal{A}$ containing three elements $\{0, 1, \#\}$. Each element represents a "move" in depth first search of T. Specifically, 0 represents a move going down to the left child of the current node, 1 represents a move going down to the right child of the current node, and # represents a move going upwards to the parent of the current node. We can encode T easily by following the depth first search. Also we can reconstruct T just by simulating a depth first search if we are given an encoded sequence of T. Since the depth first search visits each nodes at most twice, the total size of the encoded sequence for T is $O(n)$ where n is the number of nodes in T. This encoding scheme extends to general trees. However, because of simplicity of presentation, we still regard $O(n \log n)$ as the size of standard representation of a tree while $O((n + m) \log n)$ for a general undirected graph throughout this paper.

2.2 LZW Compression Algorithm and LZW Tree

Data compression algorithms try to encode a given input in a more efficient way so that the storage of the input takes less space and transmitting the input takes less time. Lempel-Ziv compression (LZ) [8, 7] is a well known algorithm that uses a dictionary to encode the original input. LZ compression parses the original sequence into subsequences by maximum prefix match and it is proven that LZ compression achieves optimal compression ratio when the size of the input goes to infinity. LZW compression is a popular version of LZ method and has been implemented as "compress" command on UNIX systems.

The main data structure used in LZW is a suffix trie. For simplicity purposes, we can view the suffix trie as a multiway tree although the actual efficient implementation of the trie involves hash tables instead of a linked list representation.

The construction of the LZW tree is as follows. We start the dictionary as an empty set. At each step, we find the longest prefix match of the remaining input to the current LZW tree. We then add the new node into the dictionary which contains the longest prefix match plus the first character of the remaining input string.

The LZW tree has many interesting properties which are critical in achieving optimality of compression. All internal nodes in LZW tree have indices in the dictionary. The shape of the LZW tree is an approximation of the actual probability distribution of the original inputs. Intuitively, each time we update the LZW tree by adding a new leaf, every prefix of that leaf which are already stored in the dictionary, appears one more time. Also, each leaf in the LZW tree has approximately the same probability of being reached when we parse the input string into a series of distinct substrings.

2.3 Stochastic Properties of Binary Tree

We define a number of stochastic properties of a binary tree in this section. For any given binary tree T, let V be the set of nodes and E the set of edges contained in T. Also let n denote the size of T (the size of E is $n-1$).

Consider the following way of generating a random binary tree with n leaves. We first expand the root node by adding two leaves. And then we expand the 2 leaves at random. At time k, we choose one of the $k-1$ leaves according to a uniform distribution and expand it until n leaves have been generated. We call the binary tree T_n we obtained a *randomly generated* tree. Note that every possible binary tree can be generated using this method. We can estimate the entropy $H(T_n)$ (the minimum number of bits needed to represent a n-node binary tree T_n using this expanding method) by writing out the recurrence relation (see page 73 [4]). The entropy rate $H(T_n)$ grows linearly with n. In other words, . there exists a constant $c > 0$ such that the number of all possible randomly generated binary trees with n leaves is less than 2^{cn}. A randomly generated tree with n leaves has less than n internal nodes. Thus the number of all possible binary trees with n nodes (counting both internal nodes and leaves) is less $2^{cn/2}$.

In our tree compression algorithm, we use breadth first search (BFS) trees to parse the original tree into subtrees. Let D_n denote the set of all BFS trees each containing n nodes. Note that D_n is only a subset of all trees with n nodes. Let $c_1 = 2^{c/2}$, we have the following lemma:

Lemma 1 *The number of distinct binary trees containing n nodes is less than $c_1 2^n$ where c_1 is a constant.*

The randomly generated tree we just described is an example of trees generated by a stochastic process. We use $\mathcal{X} = \{X_1, X_2, ...\}$ to denote a stochastic process which generates binary trees in a breadth first search order according to a certain probability distribution. The probability space of $\mathcal{X}$ is defined as $(x_1, x_2, ..., x_n)$ for

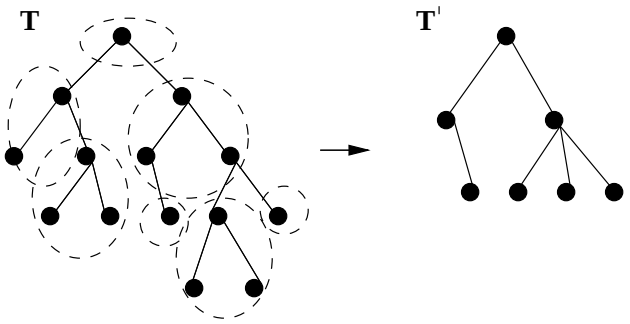

Figure 1: Parsing a tree via breadth first search.

$n = 1, 2...$ where $(x_1, x_2, ..., x_i) \in D_i$ and D_i is the set of all distinct BFS trees with i nodes. We call $\mathcal{X}$ is a *stationary* process if the joint probability distribution of any subset of the sequence is invariant with respect to shifts in the time index, i.e.,

$$Pr\{(X_1, X_2, ..., X_n) = (x_1, x_2, ..., x_n)\} = Pr\{(X_{1+k}, X_{2+k}, ..., X_{n+k}) = (x_1, x_2, ..., x_n)\} \quad (1)$$

for every time shift k and for all $(x_1, x_2, ..., x_n) \in \mathcal{X}$. In the case of tree generation, at each time index k, $\mathcal{X}$ starts to generates new trees from a leaf according to the same probability distribution as it starts at time index 0. In other words, if we cut any edge in the binary tree generated by $\mathcal{X}$, the subtree below this edge has the same probability distribution as the original tree.

We define *entropy rate* $H(\mathcal{X})$ of a stochastic process $\mathcal{X}$ as the follows:

$$H(\mathcal{X}) = \lim_{n \to \infty} \frac{1}{n} H(X_1, X_2, ..., X_n) \quad (2)$$

For a stationary stochastic process $\mathcal{X}$, let ρ_{opt} denote the optimal compression ratio for any sequence it generates, the following holds [4]:

$$H(\mathcal{X}) = \frac{1}{\rho_{opt}} \quad (3)$$

3 Our Tree-Compress Algorithm

We describe our tree-compress algorithm in the case of binary trees. It is readily extendible to general trees. The tree compression algorithm proceeds in a similar fashion as LZW data compression. The dictionary D now contains indices each pointing to a BFS (breadth first search) tree. Initially D only contains a single index which points to a BFS tree consisting only one node. We proceed by traversing the input tree T by a breadth first search. In addition, we use a queue structure S to keep track of the traversal. As in LZW compression, we first parse T into distinct subtrees and construct a dictionary D which contains BFS subtrees. Each time we visit a new node $\alpha = x_i$, we find the maximum match of the subtree under α to the BFS trees already stored in D. Let subtree $match(\alpha)$ be the maximum match. If the whole

subtree under α is already stored in D, we skip the subtree and continue parsing in breadth-first order. If $match(\alpha)$ is not equal to the whole subtree of α, we add one more index to D which points to a new BFS tree containing the maximum match plus the next node in breadth-first search order x_{i+1}. We then update the number of visits to every BFS subtrees in D. Then we repeat the process in a breadth first search manner, i.e., we parse one subtree from the leftmost subtree and break it into subtrees, then work on the second leftmost subtree and so on until the whole tree is parsed. The queue S is used to keep track of the roots of the subtrees.

After we parse T and construct D, we retain only a subset of BFS subtrees in D to compress the tree which ensures a distinct parsing of T. This is accomplished by removing all the trees with size bigger than $k = \sqrt{n}$ from D and identify all the BFS trees which is not a prefix tree of any other in D. We calculate the probability of each BFS tree remaining in D by using the counter associated with it. Specifically, let n be the sum of all the counters associated with trees in D and c_v be the number of visits of BFS subtree v in D, then the probability p_v of v is calculated as c_v/n. We then assign Huffman codes [4] to each subtree according to the probability distribution of these trees. Using this set of subtrees, we reparse T into a group of BFS subtrees and construct a tree T' which is a compressed form of T where each node in T' contains the Huffman code for the corresponding subtree in T.

It remains to describe how to compress the edges. Given D, The edges in T can be classified into two kinds: *internal edges* and *bridging edges*. Internal edges are those edges in T which are contained in one of the parsed subtrees of D. They are readily represented by the subtree which it belongs to. Bridging edge is an edge connecting two different parsed subtrees in D. Assume e is an bridging edge. We represent e by encodings of v_1 and v_2 which are the two nodes in T that e connects. To encode v_1, we first find the subtree i that v_1 belongs to after parsing. Then we find the position j of v_1 in the left-to-right order of the leaves of this subtree. We then encode e as $i\#j$ where $\#$ is a separator.

Tree-Compress (T; D, T')
Input a tree T generated by a stationary ergodic stochastic process;

Step 1(Initialization)
Initialize D to contain a single index representing a BFS tree
  containing only one node;
Initialize a queue S to contain roots of subtrees traversed in BFS;
Step 2(Parsing)
Traverse T in breadth first search order
While (search not completed) do
  Let α be the current node in T we are visiting
  Starting from α; find the maximum match $match(\alpha)$ in D for the subtree below α ($subtree(\alpha)$);
  If $match(\alpha) = subtree(\alpha)$,
    update the counters for $subtree(\alpha)$ in D;
    insert all roots of the subtrees below $subtree(\alpha)$ into S;
    let α be the next root of a subtree from S;

Else
form a BFS tree by adding the next node in breadth first search of T to $match(\alpha)$;
add this new BFS tree into D;
update the counters for all the prefix trees of this new BFS tree in D;
insert all roots of the subtrees into S after cutting the new BFS tree from T;
pop a root of a subtree from S and let it be α;

Step 3(Encoding)
Remove all trees in D with size bigger than $k = \sqrt{n}$;
Retain only those trees in D which are not prefix tree of any other tree;
Calculate the probability of occurrences for each tree in D and assign corresponding Huffman code to it;
Reparse T using D and encode the subtrees by assigning dictionary code;
Identify and encode bridging edges;

Output a dictionary D containing a set of distinct BFS trees and a compressed tree T'.

3.1 Proof of Correctness of Tree Compression

We show that the tree compression algorithm compresses an binary tree T into a compressed form T' with smaller size. Recall that we originally represent nodes of T by $\log n$ bits each.

Assume that e is an edge in T. If e is an internal edge, e is contained in a parsed subtree of T. The node in T' which represents this subtree already encodes e so no extra space is needed for encoding e in T'.

If e is a bridging edge in T, e is encoded by a concatenation of two binary numbers. Assume that v_1 and v_2 are the two nodes in G that e connects and v_2 is below v_1. Let T_1 be the parsed subtree that v_1 belongs to and T_2 be the parsed subtree that v_2 belongs to. Note that v_1 is a leave of T_1 and v_2 is the root of T_2. The edge e' in T' contains the subtree number in the dictionary for T_2 and the position of the v_1 in the left-to-right order of the leaves of T_1. The node number for T_2 takes at most $1/2 \log n$ bits since the total number of the nodes in T_2 is bounded by $\sqrt{n}$. The total number of leaves in T_1 is also bounded by $k = \sqrt{n}$. Thus the size of e' is bounded by $1/2 \log n + 1/2 \log n = \log n$ which is the size of e. Thus the encoded form of the bridging edge is of the same length as the original form.

3.2 Proof of Optimality of Tree Compression

We prove that our tree compression algorithm achieves optimal compression, i.e. the dictionary used to parse the tree is optimally selected so that the compression ratio $\rho = 1/H(\mathcal{X})$ where $H(\mathcal{X})$ is the entropy rate of a stationary ergodic source $\mathcal{X}$ from which the binary tree is generated.

Let $\mathcal{X} = (X_1, X_2, ...)$ be a stationary ergodic process that generates the binary trees in depth first search order. Let $P(x_1, x_2, ..., x_n)$ be the underlying probability

mass function for $\mathcal{X}$. For a fixed integer k, we define the kth order Markov approximation to P as

$$Q_k(x_1, x_2..., x_n) = \prod_{j=1}^{n} P(x_j | x_{j-k}^{j-1}), \tag{4}$$

where $x_i^j = (x_i, x_{i+1}, ..., x_j), i \leq j$. We will show that the algorithm is optimal when $n \to \infty$.

Theorem 1 *The tree compression algorithm is optimal when $n \to \infty$ where n is the size of the tree T which is generated by a stationary ergodic source.*

Proof: We approximate the stationary ergodic source $\mathcal{X}$ by k-th order Markov source Q_k where the probability of the next state is only dependent on the previous k states. When $k \to \infty$, $Q_k \to \mathcal{X}$ in the limit. In the construction of the final dictionary D' in our compression algorithm, we estimate the probability distribution of the leaves above k-th level. Let v be an arbitrary leaf in the dictionary and p_v be the probability that v appears in T. Let e_v be the estimation of the probability of v in Q_k. We show that for any given $\epsilon > 0$, $Prob(|p_v - e_v| \leq \epsilon) \to 1$ when the size of the tree $n \to \infty$. Also as $n \to \infty$, $k = \sqrt{n} \to \infty$ and $Q_k \to \mathcal{X}$ in the limit.

Recall that each subtree in T has the same probability distribution. Therefore the number of occurrences of v after we parse $n' < n$ times after v appears in the dictionary. Assume that v has a probability p_v in Q_k, the number of occurrences U of v in n' parses forms a binomial distribution with probability p_v.

Applying Chernoff bounds to the binomial distribution, we obtain the following,

$$Prob(U \geq (1+\epsilon)p_v n') \leq \frac{e^{-\epsilon^2 p_v n'}}{2} = \frac{1}{{n'}^{\Omega(1)}} \tag{5}$$

and

$$Prob(U \leq (1-\epsilon)p_v n') \leq \frac{e^{-\epsilon^2 p_v n'}}{3} = \frac{1}{{n'}^{\Omega(1)}} \tag{6}$$

for any given $\epsilon > 0$. As $e_v = U/n'$, we have $Prob(|e_v - p_v| \leq \epsilon) \to 1$ as $n' \to \infty$.

As the Huffman coding based on the probability distribution of the leaves at k-th level is optimal for encoding the source Q_k, our dictionary achieves optimal compression in the limit when the size of the input tree T grows to infinity. □

4 Compression of Graphs

Our algorithm for compressing a given undirect connected graph G works in two steps. First we traverse G using breadth first search to obtain a BFS tree T and then apply the tree-compress algorithm to construct a dictionary D and a compressed form T' of T. We then traverse G again to identify all back edges (see definitions in [2]) and encode them using the similar technique for encoding bridging edges in tree

compression algorithm. The edges of G can be classified as *internal edges*, *bridging edges* and *back edges*. The first two kinds of edges are contained in the BFS tree of G and are compressed in the tree compression algorithm. The back edges are compressed in the second traversal using the same technique of compressing bridging edges in the tree compression algorithm. However, the size of compressed form of a back edge may exceed the size of the original encoding. More specifically, if e is a back edge connecting v_1 and v_2 in the original graph, let T_1 and T_2 be the subtrees in the dictionary which contains v_1 and v_2 respetively. The encoded form e' of e is stored in the node in G' representing T_1 and e' consists binary numbers representing T_2, v_1 and v_2 (the pre-order for v_1 in T_1 and v_2 in T_2). We expect that our algorithm is efficient when the number of back edges is small comparing to the total number of internal and bridging edges thus the loss in compressing back edges is well compensated by the gain in compressing the other edges.

Graph-Compress (G; D, G')
Input an undirected graph G;

Step 1
Traverse G using breadth first search and construct BFS tree T for G;
Step 2
Apply Tree-Compress Algorithm on T and achieve dictionary D and compressed tree T';
Step 3
Traverse G again using breadth first search;
Compress back edges using similar strategy of compressing
bridging edges in Tree-Compress Algorithm;
Combine T' and compressed back edges to form compressed graph G';
Output a dictionary D containing a set of distinct trees and a compressed graph G'.

For a directed acyclic connected graphs (DAG), our algorithm above works similarly except that the depth first search now follows the direction of the edge. We encode the cross edges using similar method of encoding back edges in the case of compressing an undirected graph. Note the compressed edges in this case are also directed edges. Our method of compressing undirected graphs and directed acyclic graphs may not always yield a compressed form with a smaller size since the back edges and cross edges may not be compressed efficiently. However, if the back and cross edges can be identified easily (e.g., when the graph is generated by randomly adding edges to an existing tree) and the number of back edges and cross edges is only a small portion of the number of edges in the original graph, we expect that our compression scheme to perform well. Furthermore, if the graph can be parsed into a series of trees (e.g., in the case of parallel-series graph), we can apply our Tree-Compress algorithm to the parsed trees and achieve optimal compression.

5 Conclusions

In programming languages such as LISP, it requires a large database organized according to some data structure (tree, undirected graph, DAG, etc). To preserve the space for storing such a database, it is desirable to have a compression scheme which compresses the original data structure into one with smaller size. This paper tries to outline a compression scheme to compress the database while keeping a similar structure. However, in most cases, the data contained in each node of the data structure may vary widely thus our compression scheme has only limited applicability in those kinds of actual databases. In other words, the compressibility of such data structure is small. Nonetheless, if the data structure has many repeated patterns (e.g. the identical subtrees in a tree structure) and the data contained in each node varies infrequently, our algorithm is proven to be efficient.

References

[1] T.C.Bell and J.G.Cleary and I.H.Witten, *Text Compression*, Prentice Hall Company, 1990.

[2] T.H.Cormen, C.E.Leiserson and R.L.Rivest, *Introduction to algorithms,* McGraw-Hill Book Company, 1990.

[3] S. Chen and J. H. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and computational geometry for bounded-entropy inputs. *34th Symposium on Foundations of Computer Science*, 104–112, 1993.

[4] T.M.Cover and J.A.Thomas. *Elements of Information Theory.* John Wiley and Sons, New York, NY, 1991.

[5] K.H. Holm. Graph matching in operational semantics and typing. In *Proceedings of Colloquium on Trees in Algebra and Programming*, pages 191–205, 1990.

[6] D.E.Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Massachusetts, 1973.

[7] V.S.Miller and M.N.Wegman, Variations on a theme by Ziv and Lempel, *Combinatorial algorithms on words,* edited by A.Apostolico and Z.Galil, 131-140, NATO ASI Series, Vol. F12., Springer-Verlag, Berlin.

[8] J.Ziv and A.Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Information Theory,* 23, 3, 337-343(1977).