Coding and Graceful Labeling of trees

SURF 2001

Virginia Vassilevska

Abstract

A function f is a graceful labeling [1] of a graph G with q edges if it is an injection from the vertices of the graph to the set of integers $\{0, 1, \ldots, q\}$ such that when an edge with vertices x and y are assigned the label $|f(x) - f(y)|$, the resulting edge labels are distinct. The Ringel-Kotzig conjecture [5] that all trees are graceful has been attacked by many mathematicians since it was first stated in the 1960s, however not much progress has been made. This SURF project involved learning as much as possible about the graceful labeling of trees and attempting to find something new about it. Further attention was given to investigating the performance of different kinds of dynamic Huffman coding algorithms, which also involve labelings of trees.

1 Introduction

One of the most widely used data compression algorithms is Huffman's algorithm, which, using a binary tree, assigns an optimal static code to each letter in a given message. One disadvantage of Huffman's algorithm is that it makes two passes over the message - one to collect the frequencies of the letters in the message and to construct the coding tree, and one to do the actual encoding. Many attempts have been made to improve Huffman's algorithm by constructing one-pass, adaptive Huffman algorithms, the best known of which are FGK, produced by Faller, Gallager and Knuth and Jeffrey Vitter's algorithm Λ [7]. Vitter shows that both FGK and Λ perform better than Huffman on messages shorter than 104 symbols. However, as the length of the message grows, both algorithms converge to Huffman, and when more than 256 symbols are used, both are worse than Huffman.

In order for the actual performance of the Huffman, FGK and Λ algorithms to be investigated, all three algorithms were implemented using the JAVA programming language. Some results obtained from these programs are given in Table 1. They show that, when it comes to small files, algorithm Λ gives the shortest code, FGK is second in this comparison and Huffman is the worst, except in the case when the files have only one or two distinct symbols where Huffman is the best (files same and same2). The version of algorithm FGK used is one which numbers the nodes of the binary tree using Vitter's implicit labeling [7]. The results obtained are entirely experimental.

In 1998 a new adaptive Huffman algorithm was created by Pigeon and Bengio [8]. It has been shown, through experiments, that this algorithm, called M, does better than Huffman, FGK and Λ, when millions of symbols are used. This algorithm would be more useful for encoding nontext files. We are still working on finding a bound on algorithm M's asymptotical performance, which would give some more insight on when it is best to use this algorithm.

All four coding algorithms mentioned above use binary trees to encode the message. Binary trees are seen in many areas of computer science and discrete mathematics and their labeling seems an important issue to be investigated. Among the different types of labelings of trees, the so called graceful labeling attracts a lot of attention in discrete mathematics research, not only because of its beauty, as the name proposes, but also because of its difficulty. Since its main problem, to prove the Ringel-Kotzig conjecture [5] , was first stated in the late 1960s, not much progress has been made.

A graceful labeling of a graph G with q edges is an injection from the vertices of G to the set S of integers $\{0, 1, \ldots, q\}$ such that when an edge with vertices x and y are assigned the label $|f(x) - f(y)|$, the resulting edge labels are distinct. A thesis, first stated by Rosa [13] and which is equivalent to the Ringel-Kotzig conjecture [14], which is how we shall from now on address the problem, states that all trees are graceful. So far it has been shown that the following types of trees are graceful: line trees, caterpillars, trees with at most 27 vertices [6], firecrackers, symmetrical trees, olive trees and some types of banana trees. A greater progress has been achieved by Koh, Rogers and Tan [3] who gave methods for combining graceful trees to yield larger graceful trees, thus proving, among other things, that complete m-ary trees are graceful, a problem stated by I.Cahit [4]. Using Koh, Rogers and Tan's methods, Burzio and Ferrarese [2] prove that the graph obtained by subdividing every edge of a graceful graph into some number of parts is also graceful. All attempts made to provide a solution to the Ringel-Kotzig conjecture work by trying to find graceful labelings of trees. In this project we also consider this method. Using the C++ programming language we have written 3 programs, which given a tree, provide a graceful labeling. One of our programs uses exhaustive search and is not useful for trees with more than 5 vertices, as it is of factorial complexity. Another program implements the algorithm used by Aldred and McKay [6] to prove that trees with less than 27 vertices are graceful. The last program uses a simulated annealing algorithm, which has shown itself to be as successful as the AIM [6] algorithm.

Aside from using the direct approach used so far, we propose a new way of looking at the graceful tree problem. Instead of trying to label all trees, we consider a set S of n points, labeled by the number $\{0, 1, \ldots, q\}$ and by adding edges connecting these points, construct all possible gracefully labeled trees on n points. Afterwards, comparing the forms of the obtained trees, we only consider the nonisomorphic ones and count them, hoping to find a connection between the number n and the number of nonisomorphic graceful trees on n vertices. The aim here is to be able to compare the number of graceful trees and the number of all trees on n vertices, t_n. A problem encountered here is that there is no known explicit formula for the number of nonisomorphic trees on n vertices. The numbers t_n can be computed recursively using Otter's formula. Using these numbers and counting by hand of graceful trees, we show that until $n = 8$, the number of graceful trees on n vertices is the same as that of nonisomorphic trees on n vertices. We are currently working on a program, which would count the graceful trees using a matrix definition of form and could give further insights on the Ringel-Kotzig conjecture.

2 Materials and Methods

All programs written for this project were composed and run on a machine in the UGCS cluster running a SUN Solaris OS. The JAVA and C++ compilers used were javac and gcc respectively. The debuggers were jdb and gdb respectively.

2.1 Coding

The code for the coding part of the project was written in the JAVA programming language as it was assumed that later on a GUI would be composed to make the coding of text easier and more user friendly. The Node.java and Tree.java files contain the code for the Node and Tree classes.

All three algorithms code the message using binary trees. Each leaf of the tree corresponds to a letter in the message. The leaf also has a weight which corresponds to the number of times the letter appears in the message. Each letter is encoded by its node's position in the tree. Starting from the start node and going down if the path to the node is to the right, a 1 is added to the code, if the path is to the left, a 0 is added. Thus, each letter gets a binary code.

Huffman's algorithm is a static two-pass coding algorithm. It goes through the message twice - once to create the coding tree and once to encode the message. FGK and Λ are dynamical one-pass coding algorithms. They go through the message only once, creating a new tree with each newly read letter. The receiver can restore the message completely from these trees.

2.1.1 The Node Class

Each Node has a number, a weight, a symbol, a left and right child, a parent and a boolean, *es*, which indicated whether the Node is a leaf node or not. There are three main constructors responsible for creating leaf nodes, zero nodes and internal nodes. The remaining two constructors are not necessary, however may prove useful for experimenting.

The functions of the Node Class are described in Appendix III [17] .

2.1.2 The Tree Class

The Tree class contains as data members two Nodes, the start node and the node with the highest numbering, three ints, the number of nodes, the smallest number of a node in the tree and the number of leaves in the tree.

The class has three contructors. One takes a Node object and makes it the start node of the tree. It then goes through the tree, numbers it and finds the smallest number of a node in the tree, setting it to the corresponding data member of the class, mind. The second constructor takes in an int and a char and creates a one node tree consisting of the given node. The third constructor creates a tree with a zero node as its start.

The functions of the Tree class are described in Appendix III [17].

2.1.3 The Huffman Algorithm

Suppose a message M has k distinct letters $a_1, a_2, \ldots, a_k$. A Huffman coding tree has k leaves in which w_j, for $1 \leq j \leq k$, is the number of times the letter a_j occurs in M. The construction of the coding tree goes forth in the following way:

- k nodes with weights $w_1, \ldots, w_k$ are created and inserted into some kind of a container (say, a vector);
- Two nodes of smallest weight are taken out of the container and joined with a common parent node of weight the sum of their weights. This new node is then inserted into the container.
- The above procedure is performed continuously until there is only one node left in the container - the start node of the coding tree.

This procedure minimizes the weighted external path length $\sum_j w_j l_j$, where l_j is the depth in the tree of the jth leaf, and thus gives an optimal length for the code.

2.1.4 Algorithms FGK and Λ

Faller [10] and Galler [11] independently developed a one-pass algorithm, which after being substantially improved by Knuth [12] , became to be known as algorithm FGK. Vitter later on improved it even further creating his Λ algorithm.

As already mentioned, both Λ and FGK are dynamical codes. They are not based on a single static tree, but on a dynamically varying Huffman tree [7]. The tree used to encode the $(t+1)$st letter in the message is one based on the information one has from the first t letters in the message. The sender and receiver of the message modify their copies of the coding tree after processing each letter. No trees are ever transmitted, as both the receiver and the sender use the same modification algorithm and start from the same starting position - a 1 node tree.

The main problem here is to find a modification algorithm which transforms a Huffman tree for the first t letters of the message to a Huffman tree for the first $t+1$ letters of the message. This modification algorithm is where Lambda and FGK differ in.

The modification algorithm of both codes depends on a numbering of the nodes in each Huffman tree. In our programs we consider a numbering based on the visual representation of the tree, the so called implicit numbering proposed by Vitter. This makes the version of FGK we are using a cross between FGK and Λ. The original FGK algorithm does not specify the numbering to such a great extent, so that there are many possible valid numberings of a Huffman tree. The FGK numbering only requires that sibling nodes have numbers which differ by 1 and that the number of the parent node of those sibling nodes is higher than any of their numbers.

The implicit numbering, on the other hand, is such that the nodes on the same level are numbered increasingly from left to right and nodes on a lower level are numbered lower than nodes on a higher level.

The modification algorithm of FGK is as follows: Before adding the new letter to the tree, we start updating the tree. We find the node which corresponds to the next letter and call it the current node. If the letter has not yet been encountered, the zero node is the current node. We then exchange this node with the node of the same weight with the highest numbering. We then make the parent of the latter node the current node and continue in the same fashion. If no exchange is possible, then the parent of the current node becomes the current node and the procedure continues until the start node (=root) is reached. When this happens, we increase the weight of the node for the next letter by 1. If this is the zero node, then we replace it by an internal node with weight 1 and children a zero node and a leaf of weight 1 and letter the next letter. The weights of the ancestors of the node with increased weight are also increased by 1 in order to turn the tree into a valid Huffman tree for the message read so far.

The modification procedure of algorithm Λ is similar. However, one more constraint is added to optimize the code. It states that in the Λ numbering, leaves of weight w have lower numbers than internal nodes of weight w. Vitter proves by analyzing the coding process that this constraint makes his algorithm better than FGK. Our experimental results confirm this statement.

2.2 Graceful Labeling

The methods used in our investigation of the problem of graceful labeling of trees include reading various papers on the subject, all of which are quoted and their results are summarized in Gallian's *A Dynamic Survey of Graph Labeling* [1]. Computer programs, written by us using three different algorithms, aided us in our labeling experiments. Two of those programs are based on algorithms designed by us, and the third one uses the algorithm AIM, which was devised by Aldred and McKay and which proved that all trees on 27 vertices or less are graceful.

The results we obtained using the above methods are described in the next section.

3 Results

3.1 Coding

The results of the comparison between the three Huffman coding algorithms are summarized in table 1. They show in essence that in most cases algorithm Λ performs better than both FGK and static Huffman, except in the case when the number of distinct symbols in the message is small, as is the case with files same and same2. 1 In the latter case the static Huffman algorithm does the best. Furthermore, when the size of the file grows, the length of the codes generated by FGK and Λ converges to the one generated by Huffman,

The performance of the three implemented algorithms was comparatively good. FGK and Λ are only a bit slower than Huffman. Using our implementations we were able to encode and decode the entire "Hitchhiker's guide to the galaxy" for under three minutes on a 700MHz processor.

As mentioned earlier in the paper, besides working on these three algorithms, we read about a newer dynamic Huffman algorithm designed for handling millions of symbols. The authors of this algorithm, Pigeon and Bengio [8], claim that when millions of symbols, instead of the usual 256, are used, algorithm M outperforms Huffman, FGK and Λ in nearly all cases. However, they do not give a bound for the performance of their algorithm the way Vitter does for the other three. We find it interesting to investigate the problem of finding a bound, as it would help us understand in which cases which algorithm should be used. So far no bound has been found.

File / Algo	Huffman			FGK			Lambda		
	cl	b/s	cpr	cl	b/s	cpr	cl	b/s	cpr
Fgk.java 5127 symb.	24954	4.86	39%	25137	4.9	38%	25040	4.88	38%
newf 199277 symb.	915121	4.59	42%	915468	4.59	42%	915386	4.59	42%
hitch 286420 symb.	1313802	4.58	42%	1314293	4.58	42%	1314185	4.58	42%
baba 20 symb.	73	3.65	54%	69	3.45	56%	62	3.1	61%
con 4 symb.	8	2.0	75%	6	1.5	81%	5	1.25	84%
same 14 symb.	14	1.0	87%	13	0.92	88%	13	0.92	88%
same2 209 symb.	314	1.5	81%	415	1.98	75%	415	1.98	75%

Table 1: Statistics for the three algorithms

cl stands for code length in symbols
b/s stands for bits per symbol
cpr stands for compression

3.2 Graceful Trees

One of the first things to notice when approaching the topic of graceful labeling is the fact that line trees are graceful. This fact is already known. An inductive proof, which we devised, goes as follows:

Theorem 1 *Line trees are graceful.*

Proof:
Consider a representation of the tree T with n vertices as a straight line going from left to right. Label the vertices going from left to right in the following way: $(1, n, 2, n-1, 3, \ldots, \lceil \frac{n}{2} \rceil + 1, \lceil \frac{n}{2} \rceil)$ if n is odd and $(1, n, 2, n-1, 3, \ldots, \frac{n}{2}, \frac{n+2}{2})$ if n is even. One can easily check that this procedure creates a graceful labeling of line trees with $1, 2$ and 3 vertices. Suppose it works for a tree with n vertices. Now relabel the tree by exchanging each label l of a vertex with the label $n-l+2$. The edge labels do not change and take all values from 1 to $n-1$. The vertex labels range from 2 to $n+1$. Then, one can add a new vertex to the left of the tree and label it with a 1, as 1 does not occur in the new labeling. The new vertex is connected then to the leftmost vertex, thus creating an edge labeled with n. Now the vertex labels range from 1 to $n+1$ and the edge labels range from 1 to n. We get a graceful labeling of a line tree with $n+1$ vertices. The proof is finished.

In experimenting with various labelings of trees, we considered writing programs to help us in this matter. Appendix II [16] contains the codes of all three programs written by us on graceful labeling.

The first program, solve, tries to find all possible graceful labelings of a given binary tree. As it uses exhaustive search, it has a factorial complexity and is not useful for $n > 4$.

The second program, amsolve, uses the algorithm designed by Aldred and McKay [6]. It uses a random number generator to pick a pair whose labels are then swapped. As described in Aldred and McKay's paper [6], the Graceful and Harmonius algorithm goes as follows:
Given a tree T and some labeling of the vertices L, let $z(T, L)$ be the number of distinct labels in L. If n is the number of the vertices, then the aim is to find L so that $z(T, L) = n - 1$. If v and w are vertices of T, let $S_w(L; v, w)$ be the labeling one gets by swapping the labels of v and w.

1. Start with any labeling.
2. If $z(T, L) = n - 1$, stop.
3. For each pair $\{v, w\}$, replace L by $L' = S_w(L; v, w)$, where $\{v, w\}$ is chosen at random from the set of all $\{v, w\}$, such that
   - $\{v, w\}$ has not been chosen in the last M times.
   - $S_w(L; v, w)$ is maximal subject to the previous constraint.
4. Repeat from 1.

The value of the paramater M varies with the size of the tree you want to label. For smaller trees $M = 10$ is sufficient. For larger trees, M should also be larger. We gracefully labeled a 64 node tree with $M = 20$. The purpose of M is to prevent the algorithm from cycling repeatedly among some small number of labelings. The larger M, the higher the probability that a graceful labeling will be found. However, the larger M, the slower the algorithm gets.

The last program, simul, uses a simulated annealing algorithm. It minimizes the number of repeating edge labels. This method may prove convenient, as one can easily add restrictions to the type of graceful labelings one is looking for. For example, one may wish to search for labelings where sibling nodes have consecutive labels. Like the other two programs, this one is also written for binary trees. Without much change one can rewrite it for trees with vertices with higher degrees.

Both simul and amsolve perform reasonably well. They were able to label a 64 node tree in less than 8 minutes. Of course, as they are using random numbers, it is possible that a solution is never found. Yet still, these algorithms are much better than the exhaustive search which could not even label a 5 node tree.

As noted in the introduction, we propose a new way of looking at the graceful tree problem. The method involves counting the nonisomorphic graceful trees that can be built from n labeled edges. Our counting reveals that the number of graceful trees that can be composed for $n < 9$ is equal to the number of nonisomorphic trees on n vertices. A problem we encountered is that there is no efficient way to find the number of nonisomorphic trees on n vertices. Formulae exist (by Otter and Polya). However, they only allow a recursive calculation, which becomes inefficient for larger values of n. Furthermore, there is no known formula for determining the number of nonisomorphic graceful trees that can be composed from n labeled vertices. The number of all possible graceful graphs on n vertices is $(n-1)!$. Excluding disconnected, nonsimple graphs and graphs with polygons, would lead to another number, which is then to be reduced by excluding trees of the same form, so that in the end we get the number of all nonisomorphic graceful trees on n vertices. If this number can be found, then one could compare with the number of nonisomorphic trees on n vertices, or with its lower bound, $\frac{n^{n-2}}{n!}$, which would prove especially useful if the Ringel-Kotzig conjecture is not true. Thus, proving or disproving the Ringel-Kotzig conjecture is equivalent to finding and comparing those numbers. Looking at the problem from this direction would allow a more efficient program algorithm of finding values of n greater than the so far known $n \leq 27$ for which all trees on n vertices are graceful.

References

[1] J. Gallian: *A Dynamic Survey of Graph Labeling*, JACM, 1997

[2] M. Burzio, G. Ferrarese: *The subdivision graph of a graceful tree is a graceful tree*, Discrete Math. 181 (1998) 275-281

[3] K.H. Koh, D.G. Rogers, T. Tan: *Two theorems on graceful trees*, Discrete Math. 25 (1979) 141-148

[4] I. Cahit: *Are all complete binary trees graceful?*, Am. Math. Monthly 83 (1976) 35-37

[5] S.W. Golomb: *How to number a graph*, in: R.C. Read, ed., Graph Theory and Computing, (Academic Press, New York, 1972) 23-27

[6] R.E. Aldred, B.D. McKay: *Graceful and harmonious labelings of trees*, preprint

[7] J.S. Vitter: *Design and Analysis of Dynamic Huffman Codes*, JACM, Vol.34, No.4, October 1987, pp.825-845

[8] S. Pigeon, Y. Bengio: *A Memory-Efficient Adaptive Huffman Coding Algorithm for Very Large Sets of Symbols*, Data Compression Conference, DCC 1998 March 10 - April 1, 1998, Snow Bird, Utah, USA, IEEE Computer Society 1998, 568

[9] F. Harary: *Note on the Polya and Otter formulas for enumerating trees*, Michigan Math. J. 3 (1956) 109-112

[10] Faller, N.: *An adaptive system for data compression.* In Record of the 7th Asimolar Conference on Circuits, Systems and Computers. 1973, pp. 593-597

[11] Gallager, R.G.: *Variations on a theme by Huffman.* IEEE Trans. Inf. Theory IT-24, 6 (Nov. 1978), 668-674.

[12] Knuth, D.E.: *Dynamic Huffman coding.* J.Algorithms 6 (1985), 163-180

[13] A.Rosa, *On certain valuations of the vertices of a graph*, in Theory of Graphs (Internat. Symposium, Rome, July, 1966, Gordon and Breach, N.Y. and Dunod Paris, (1967), 349-355.

[14] G.Ringel, Problem 25, in *Theory of Graphs and its Applications*, Proc. Symposium Smolenice 1263, Prague, 1963

[15] Virginia Vassilevska: Appendix I (Node.java, Tree.java, Trys.java, Fgk.java, Lambda.java), SURF 2001

[16] Virginia Vassilevska, Kaloyan Penev: Appendix II (solve.cpp, amsolve.cpp, simul.cpp, io.cpp, bintree.h, io.h), SURF 2001

[17] Virginia Vassilevska: Appendix III (explanations of the Node and Tree functions)