Better Huffman Coding via Genetic Algorithm

Cody Boisclair
Artificial Intelligence Center
University of Georgia
Athens, Georgia 30605
United States of America
Email: codemanb@uga.edu
Telephone: +1 706 389–6188

Markus Wagner
Artificial Intelligence Research Group
University of Koblenz-Landau
56016 Koblenz, Germany
Email: wagnermar@uni-koblenz.de
Telephone: +49 (0) 1577–4506194

***Abstract*—We present an approach to compress arbitrary files using a Huffman-like prefix-free code generated through the use of a genetic algorithm, thus requiring no prior knowledge of substring frequencies in the original file. This approach also enables multiple-character substrings to be encoded.**

We demonstrate, through testing on various different formats of real-world data, that in some domains, there is some significant advantage to using this genetic approach over the traditional Huffman algorithm and other existing compression methods.

***Index Terms*—Genetic algorithm, data structures, application, Huffman coding.**

I. Introduction

A. Background

Huffman coding, as initially described by David Huffman [2], is a particular method of compressing data through the use of a code table with encodings of variable lengths. A Huffman code is an optimum, or minimum-redundancy, code, which means that messages which occur with greater probability have shorter encodings; in addition, it is prefix-free, meaning that no code in the table may be the beginning part of any other code.

Huffman describes an algorithm which can be used to generate a binary Huffman code from a collection of messages, or strings, ordered by probability. To generate a code, one starts with a collection of all messages in order of probability. The two least probable messages are removed from the collection and combined into a "composite message," with probability equal to the sum of the messages comprising it. This process is repeated until there is only a single composite message left in the collection, with a probability of 1; that composite message represents the entire Huffman code. This is easily converted to a tree-based approach, in which the initial messages are represented as leaf nodes, each edge represents a digit 0 or 1 in the encoding, and "composite messages" are subtrees created by assigning a common parent to the merged messages.

B. Research Goals

We wanted to see whether it was possible to generate a relatively efficient prefix-free code without knowing in advance the probabilities of substrings in a source file, by means of a genetic algorithm.

There is the definite possibility that substrings of several bytes in length may be more probable as a whole than the individual bytes that comprise them—for instance, 'qu' is likely to be more common in English text than 'q' alone. Determining the probabilities of these longer substrings for a given file, however, can be quite costly. To calculate the probabilities of single-byte substrings, one must simply scan through the file and increment one of 256 counters for each byte value found. Considering both single-byte and two-byte strings, there must be $256^2 + 256 = 65\,792$ counters used; additionally, at each byte in the file, two different comparisons must be made. Adding three-byte strings to the probability table requires the use of $256^3 + 256^2 + 256 = 16\,843\,008$ different counters, with three different comparisons made at each byte of the file.

C. Related Work

This is not the first experiment that has been done to determine whether evolutionary computation may be applicable to Huffman-style compression.

Üçoluk and Toroslu [6] demonstrated a system to determine the optimal Huffman coding for a corpus of Turkish text via a genetic algorithm. In Üçoluk and Toroslu's experiment, the set of all possible syllables was known ahead of time, and the genetic algorithm was simply used to *select* the most efficient set of syllables to use in the encoding; this is in contrast to our experiment, which attempts to *originate* the substrings via evolution.

Oroumchian et al [4] used a genetic algorithm to compress a single file of Persian text. At first, they computed the frequencies of n-grams (i.e., n-character strings) with a maximum length of five; they then used a genetic algorithm to trade off the amount of n-grams versus filesize.

Another evolutionary approach using unspecified values was taken by Polian et al [5]. A string of bits can be represented as a collection of fixed-length 'matching vectors' containing the bits 0, 1, and unspecified; each matching vector is assigned a prefix-free code, and so a string may be represented as a collection of matching-vector codings followed by the values of the unspecified locations. In Polian's experiment, a set of matching vectors is generated by a genetic algorithm and then used to compress the data, with more optimal compression producing better fitness for that set of vectors.

II. Experimental Methodology

A. Experimental Details

Our genetic algorithm uses a genome containing a binary tree structure, with each genotype representing a particular prefix-free encoding. The initial population consists of a set of trees, each containing all of the 256 possible bytes as leaf nodes, with each byte having an encoding of equal length.

The genetic operators used were inspired by, but are not identical to, those used in genetic programming. One mutation, which we have named 'swap', is identical to that used in genetic programming: two nodes within the tree, each of which may be either a leaf or an internal node, are swapped with one another, so long as one node is not an ancestor of the other. Swapping a node and its ancestor is not allowed. An example of a swap mutation, in this case between a leaf node and an internal node, can be seen in Figures 1 and 2.

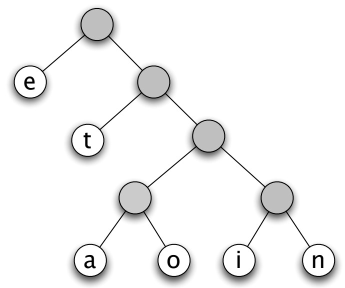

Fig. 1. A simplified prefix-free tree before the 'swap' mutation has been applied.

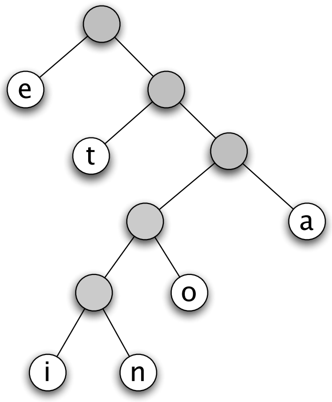

Fig. 2. The tree from Figure 1 after the 'combine' mutation has been applied, with the node 'a' swapped with the parent of nodes 'i' and 'n'.

Another mutation, known as 'combine', has been added to provide the functionality of creating the longer substrings described above. Here, two leaf nodes are randomly selected for combination; the strings in them are combined into a new string, and a new node is created to contain that string. One of the two selected leaf nodes is randomly chosen, brought down a level, and the new node is made its sibling. Figures 3 and 4 are an illustration of one particular case of the 'combine' mutation.

No crossover operator is used in this particular genetic algorithm, unlike in genetic programming; by the nature of the trees used in this algorithm, any crossover would have significant potential to cause some strings in the tree to occur twice while others disappeared entirely.

The fitness of a chromosome is determined by the length of a file, in bits, as compressed using the encoding it represents; in this experiment, therefore, fitness is to be minimized. This length is the sum of:

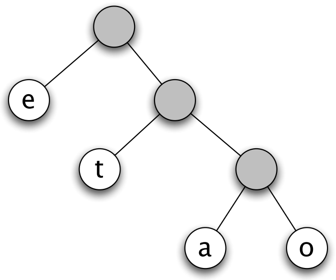

Fig. 3. A simplified prefix-free tree before the 'combine' mutation has been applied.

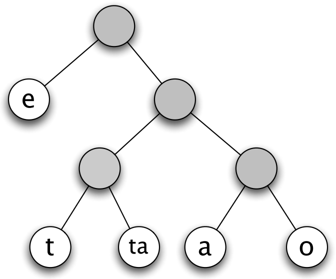

Fig. 4. The tree from Figure 3 after the 'combine' mutation has been applied, with the nodes 't' and 'a' chosen for combination and 't' chosen to be the new node's sibling.

- $2n-1$ bits to represent the tree structure, for n leaf nodes. A prefix-free encoding tree can be represented in its entirety by taking a preorder traversal of the tree and identifying whether each of the nodes visited is a leaf node or an internal node; as it is a full binary tree, there is always one fewer internal node than there are leaf nodes.
- $8k$ bits for each uncompressed string in the encoding, where k is the length of the string in bytes.
- The length of the compressed data, as determined by looking up each character in the code table and summing the lengths of their respective encodings.

This algorithm was implemented using the JGAP (Java Genetic Algorithms Package) library [1]. A 100% probability was used for mutation; in each mutation, there was a 50% probability as to whether 'swap' or 'combine' would be used. Only the fittest 10% of individuals after mutation were selected to move on to the next generation; any remaining individuals in the new generation were duplicates of those selected chromosomes.

TABLE I
TEST SUITE

File	**Original Size in Bit**
$Genome_1$	322 416
$Genome_2$	9 421 872
$Genome_3$	103 071 904
$Image_{1bit}$	5 947 888
$Image_{4bit}$	23 790 512
$Image_{8bit}$	47 587 760
Log	1 857 560
Exe_1	380 512
Exe_2	2 009 472
Exe_3	8 282 112
Exe_4	160 465 216

Population size and number of generations varied from experiment to experiment, and will be discussed in detail in the experimental analysis below.

B. Test Suite

A total of eleven test files were used – see Table I. The first three files are specially selected from the genome domain, in order to limit the size of the alphabet, thus making human observations during the evolutionary process easier. These files are samples of chromosome regions of the human genome and are available from the National Center for Biotechnology Information [3].

The next three files are versions of a landscape photograph with an original color depth of 24 bit. Limiting the color depth to at most 8 bit implicitly limits the actual alphabet.

The remaining five files were selected from a pool of real-life data: four different executable files, and one log file containing timestamps and English text.

In contrast to the experiments on the genome data, we did not assume any kind of limitations on the size of the alphabet for the rest of the test suite.

III. TEST RESULTS

In the following sections, we will use the term *space saving* to quantify the reduction in size relative to the uncompressed size:

$$SpaceSavings = 1 - \frac{CompressedSize}{UncompressedSize} \quad (1)$$

The higher the value for the space saving is, the better the compression performs.

A. Genome Data

The balanced binary tree with four leaves resulted in savings of 75 % for each of the three files. In fact, this was expected to be the case, because the four 8-bit-symbols A, C, G and T can be encoded using only 2 bits, i.e. 00, 01, 10 and 11.

When we started with the tree containing only four code points, we did not observe any growth in the tree. The theoretical explanation for this: In a balanced tree with four leaves, the space saving is 1 - (2 bits / 8 bits) = 75%. As soon as a new leaf is introduced, a new internal node has to be inserted, resulting in new and worse space saving for one of the old leaves of 1 - (3 bits / 8 bits) = 62.5%.

1) Short-time runs: Each short-time run was stopped after 100 generations, each containing 60 individuals, resulting in a total of 6 000 evaluations. Based on $Genome_1$, 30 runs were done, which were divided into two sets of 15 runs each. Each run took about 40 seconds.[1]

For the first 15 runs, the trees were initialized with the complete set of all 16 2-character strings that were likely to occur in the file. The average space saving of 75.3% is an almost negligible improvement over the 75% of the very first runs, even though one must keep in mind that the tree representation is part of the file as well. The smallest size of the compressed file was 79 731 bits. The standard deviation over all best individuals of the runs is only 136 bits.

In the next 15 runs, the four single-character code points were added to the initial individuals, resulting in trees with 20 code points. These code points were injected to provoke productive concatenations resulting in three-character strings, which would have been impossible with two-character strings alone. As an effect of this bloating, the average fitness of the initial population was even worse, but on average, a space saving of 75.0% was reached, resulting in a minimum file size of 80 682 bits and a standard deviation of 292 bits.

The runs were repeated for $Genome_2$ and $Genome_3$. The performance of the evolved trees is compared in Table II with ZIP compression. In the table, 'GA Savings' refers to space saving achieved by the GA-generated trees, while 'ZIP Savings' refers to those achieved by the ZIP method using the program WinAce [2] and the setting "normal compression".

TABLE II
PERFORMANCE OF SPECIALIZED TREES ON THE GENOME DATA

File	GA Savings	ZIP Savings
$Genome_1$	75.3 %	65.1 %
$Genome_2$	75.2 %	66.4 %
$Genome_3$	75.3 %	66.0 %

2) Long-time runs: In the next test set, we used $Genome_1$ again to let the Huffman tree evolve, but the population was enlarged to 600 individuals, while keeping the limit of 100 generations, resulting in 60 000 evaluations per run. Every run took about seven minutes. Again, 15 repetitions were done for the versions using 16 initial code points and 20 initial code points.

For the first setup, no new minimal file size was found. For the second setup, a new minimal file size of 80 004 bits was found. In both setups, the standard deviations decreased to 5 bits and 209 bits.

It is remarkable that the best solution with the fitness of 79 731 had no code points in its tree with a length greater than 2. This means that the tree was only an adjustment to the relative frequencies of the 2-character strings in $Genome_1$.

The longest substring found in the 12 best-so-far solutions was five characters long and used an encoding of 8 bits, thus resulting in a space saving of 80% for that specific string. In the fittest tree of each run, the longest encodings belonged to the single-character code points.

If the tree is balanced and only the four characters appearing in genome data are used, the space savings are 75% for that data. The approaches using trees which contained all 16 of the 2-character combinations, with or without the addition of the four single-character code points, showed no significant improvement in compression.

To see how well the specialized trees from the previous experiments could be used in general, we compressed the other files with the evolved trees.

[1]The underlying system is a 1.6 GHz Dualcore notebook with 2GB RAM and the Java Runtime Environment 1.6.

[2]The used test-version of WinAce 2.69 can be downloaded from http://www.winace.com/.

TABLE III
PERFORMANCE ON THE IMAGE TEST SUITE

File	GA Saving	GIF Saving	JPEG Saving
$Image_{1bit}$	81.4 %	28.8 %	-42.4 %
$Image_{4bit}$	47.7 %	76.2 %	49.4 %
$Image_{8bit}$	32.4 %	65.7 %	72.0 %

The evolved tree for $Image_{1bit}$ resulted in space savings of 19.5 % for $Image_{4bit}$ and 8.9 % for $Image_{8bit}$. When comparing these values to 47.7 % and 32.4 % that were achieved using specialized tree, the generalization was not as big as we expected it to be.

B. Image Data

In the next set of runs, we used the set of images of a maximum color depth of 8 bit and compared the performance of the GA to state-of-the-art image compression methods. The Graphics Interchange Format (GIF) is a wide-spread graphics format and has a maximum color depth of 8 bit. It stores the defined colors in a palette and the files are compressed using the Lempel-Ziv-Welch compression technique. In contrast to GIF, which can be considered as a lossless compression algorithm in our case, the JPEG Interchange Format is a lossy file format using a color depth of 24 bit. We used a high-quality setting of 90% for the JPEG compression. For each of the three files, specialized trees with an initial set of 256 leaves were evolved using 6 000 evaluations. In Table III, the results of evolved trees for the images are shown. On average, the GA performs quite well in the lower color depths in comparison to the other methods. For the first image, it clearly outperforms the others, but it does not reach the savings achieved by the other methods for the third image. Surprisingly, the JPEG compression of the image with the lowest color depth resulted in a blow-up of the file size by about 40%.

C. Real-World Data

In our last tests, we used the set of executable files to study the compression behavior with trees containing all 256 possible 8-bit ASCII characters. Again, the achieved space savings were compared to ratios that can be achieved using ZIP compression.

TABLE IV
PERFORMANCE OF THE SPECIALIZED TREE ON THE COMPLETE TEST SUITE

File	GA Savings	ZIP Savings
$Genome_1$	6.1 %	65.6 %
$Genome_2$	6.2 %	66.4 %
$Genome_3$	6.4 %	66.0 %
$Image_{1bit}$	5.2 %	83.7 %
$Image_{4bit}$	6.1 %	73.1 %
$Image_{8bit}$	6.7 %	65.8 %
Log	-10.1 %	85.8 %
Exe_1	11.6 %	56.2 %
Exe_2	12.2 %	43.7 %
Exe_3	8.6 %	51.6 %
Exe_4	9.2 %	47.1 %

First, we performed 15 runs using Exe_1 and let each run for 60 000 total evaluations, which took about 21 minutes for every run. The biggest saving achieved was 18.7%, resulting in a file size of 309 301 bits, while the standard deviation was again very low with 1 917 bits.

In a modification of that experiment, Log was compressed with space savings of at most 21.6 %, while ZIP compression achieves 81.2 %.

In the final run, we used Exe_2 to produce a tree for the compression and applied that tree to all files from the test suite. Again, we used a run time of 60 000 evaluations, which took about 48 minutes. Table IV presents the results of applying the Exe_2-based tree to all files.

The evolution of the tree added 24 code points with a length of two bytes or more to the existing 256 code points of one byte each. The longest codepoint found in the tree encoded 48 bits using only 8 bits, while, on the other hand, a 13-bit long encoding for a single 8-bit character was found as well.

The performance of the generated tree based on Exe_2 was constant over the other executable files. This suggests that this particular tree can be classified as specialized for executable files. The worse compression of the log file and the picture using this tree supports this statement as well.

Considering the results for the genome data, it is remarkable that even though the ZIP compression outperformed the unoptimized tree in this application, a specialized tree from the first short-run test would outperform ZIP with space savings of at most 75%.

IV. Conclusions and Future Work

We have developed a Genetic Algorithm that is capable of extending the standard Huffman coding to multiple-character encodings for better space saving.

Several times, we observed the productive concatenation of strings as well as reasonable ordering of the code points in the tree. Measured over all comparable runs, the standard deviation was always very low, with values consistently less than 1% of the average.

Once the trees have been generated, the compression of a file can be done very quickly in a single pass. Using a domain-specific tree generated in advance by evolution can prove faster than most other compression techniques, but we must not neglect the facts that finding the tree may take unacceptably long and that the compression is fairly poor in comparison to existing compression software.

In some of the tested domains, existing compression algorithms outperform our technique in time consumption and space saving. For the compression of data that uses a limited alphabet, such as genome data and images of low color depth, we recommend the use of our genetic approach.

Future work will include the application of multi-objective optimization techniques in order to improve the performance of single trees over an entire domain.

References

[1] D.-Y. Chen, T.-R. Chuang, and S.-C. Tsai. JGAP: a java-based graph algorithms platform. *Softw, Pract. Exper*, 31(7):615–635, 2001.

[2] D. Huffman. A method for construction of minimum-redundancy codes. *Proc. of IRE*, 40(9):1098–1101, September 1952.

[3] National Center for Biotechnology Information. Website. http://www.ncbi.nlm.nih.gov/mapview/seq_reg.cgi?taxid=9606&chr=1&from=1&to=247249719.

[4] F. Oroumchian, E. Darrudi, F. Taghiyareh, and N. Angoshtari. Experiments with persian text compression for web. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 478–479, New York, NY, USA, 2004. ACM Press.

[5] I. Polian, A. Czutro, and B. Becker. Evolutionary optimization in code-based test compression. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1124–1129, Washington, DC, USA, 2005. IEEE Computer Society.

[6] G. Ucoluk and I. H. Toroslu. A genetic algorithm approach for verification of the syllable-based text compression technique. *Journal of Information Science*, 23(5):365–372, 1997.