A Fast Counting Data Compression Algorithm

Warren Hunt, Advisor: Danny Sleator

May 2, 2004

1 Abstract

I propose a novel class of compression Algorithms based on counting that will provide compression ratios and speeds similar to the ratios achieved by block sorting methods as well as many additional useful features. Counting methods provide flexible tradeoff between speed and compression ratio, in many cases, fast searching of compressed files, and merging of compressed files into (possibly smaller) archives. In addition, counting methods provide an improvement over statistical methods and offer a simple algorithm that, in the 0th order, is at least as good as Huffman coding. Finally I will provide a working implementation of a specific algorithm in this class, present a few of its features, and compare its performance with other algorithms commonly used today.

2 Definitions

First, it will be useful to provide a many definitions that will clarify meaning in the rest of this paper.

Definition 2.1 Alphabet: *I will define an alphabet as a finite set of characters which could appear in a file. The standard alphabet I will use will consist of 256 characters, the configurations of a byte. However, these algorithms will work with arbitrary alphabets over a finite number of characters. I will denote an alphabet by A.*

Note: *The size of the alphabet is simply the number of characters it contains. I will denote the size of the alphabet by $|A|$.*

Definition 2.2 Character: *A character is simply any element in an alphabet. I will represent an arbitrary character in an alphabet by x. This will appear in such statements as $\forall x \in A$.*

Definition 2.3 File: *I will define a file as a finite string of arbitrary characters from a given alphabet. I will denote a file as F and the ith character of F as $F(i)$. The length of the file will be denoted $|F|$.*

Note: The file is indexed such that first character of the file is $F(0)$ and the last character is $F(|F| - 1)$. I will consider files to be cyclical. By this, I mean $F(i)$ is defined for all $i \in \mathbb{Z}$ and $F(i) := F(imod|F|)$. Infinite streams of data may be quantized into packets for compression by these algorithms.

Definition 2.4 Pattern: *A pattern P is a finite set of $(< integer >, < character >)$ pairs under the constraint that no two integers in the set are equal. The pattern implies that each given $< character >$ occurs at offset $< integer >$ for all pairs. The length of the pattern is the size of the pattern set and will be denoted $|P|$. The notation $P(i)$ refers the character paired with integer i.*

Definition 2.5 Collection: *A collection of patterns, indexed by a finite set of integers S, is a set containing every pattern such that the set of the integers in each pattern is the same as the given set S. The length of a collection is the size of the set S. The size of, or number of pairs contained within, a collection is $|A|^{length(S)}$.*

Definition 2.6 Count: *A count, indexed by a file paired with a collection, is a function which takes a pattern in its given collection and returns a non-negative integer. The value returned is the number times that the argument pattern occurs in the given file. I will denote a count as (F, S) and given a pattern $P \in S$, $(F, S)(P)$ is the number of occurrences of P in F. E.g. $(F, \{0\})(\{(0, `a`)\})$ would give us the number of occurrences of the character ‘a‘ in the file.*

Definition 2.7 Partial Count: *I will denote a partial count as a count with at least one restriction to its domain. I will use the notation $(F, S)_{\{(index, character)\}}$ to denote the restriction that $< character >$ appears at $< index >$ when the function is applied. The length of the argument required is decreased once by each restriction. E.g. $(F, \{0, 1\})_{\{(0, `a`)\}}(\{(1, `b`)\})$ returns the number of times the pattern "ab" occurs in the file.*

Note: It is important to notice that some counts contain enough information to derive others. Specifically, if the set of integers indexing one count is a subset of the set of integers indexing another count, then the first count is obtainable from the second by summing. E.g. given $(F, \{0, 1\})$ we may obtain $(F, \{0\})$ by $(F, \{0\}) = \sum_{x \in A}(F, \{0, 1\})(\{(1, x)\})$. **Note:** It is also important to clarify the meaning of $(F, \{\})$. $(F, \{\})$ is the empty count and only has one valid argument, the empty set. $(F, \{\})(\{\})$ returns the length of the file. This set is defined for inductive reasons.

3 Introduction/Motivation

Although there are many algorithms on the market, there is always room for improvements in compression, and although drives are getting bigger at a phenomenal rate, compression will always be useful in fixed size media and anywhere were bandwidth is limited. Algorithms may also be used to provide useful features in addition to reducing the size of files. The original motivation for this class of algorithms comes from the benefits of statistical compression algorithms over pattern matching or block sorting ones. A counting method is a more precise statistical method (instead of knowing the relative frequencies for each pattern, we know the exact frequencies) and gains all the benefits of being such. Statistical analysis of data streams has been done since the 30s, when Shannon provided a law that gives a theoretical bound for how much information is actually present in a set of data based on its statistical characteristics. Slow statistical algorithms have provided results close to these bounds but were never practical next to the much faster LZW and BW algorithms. Counting methods can be implemented to be much faster than traditional statistical algorithms.

Statistical methods also have an advantage in that, depending on the order of the statistical information in the file header, the ratio between speed and compression ratios can be varied widely. Pattern matching algorithms (LZW) and block sorting algorithms (BW) in general only have two settings, on and off.

Additional motivation for counting methods is provided by the fact that knowing the exact frequencies for each pattern in the file provides as good as or better compression of that file than can be provided by knowing only the relative frequencies of the patterns. This implies that a list of the frequencies for each byte in a file provides as good or better compression than the Huffman algorithm on that file. I will prove this in the next section. An algorithm more optimal than the Huffman algorithm may be implemented as follows: We know the exact count for each byte in the file and therefore may construct a Huffman tree based on those counts and compress the first byte (just like normal Huffman). However, for the next (and all subsequent bytes) we may decrement the count for the byte last compressed and reconstruct the optimal Huffman tree for the remainder of the file. Each byte is compressed by a more optimal Huffman tree.

3.1 Features

There are also many other attractive features to compression using counting methods. The first of which involves compressing multiple files. In every other compression algorithm I've encountered, when a file is added to an archive, it is effectively just compressed normally and appended to the end. With counting methods, files can be easily merged, simply by taking the two counts and adding them together. If the counts are similar, then representing the sum of the two counts takes fewer bytes than representing each independently. Files

merged in this way still maintain individual identity by having different starting indices. (Starting indices are explained later) File merging may provide a substantial benefit when compressing lots of files. Another attractive feature of counting methods is the fact that files compressed in this do not need necessarily be fully uncompressed before it may be searched. Although this is not true of all algorithms in this class, it is of the one I chose to implement and provides a notable benefit. This will be covered in detail in a the results section of the implementation. The last advantage I will mention here comes from the fact that the collections do not require their indexing integers to be sequential. Thus for storing files such as bitmaps which have information stored in the clusters of size three, patterns indexed 0, 3, 6 and so forth are much more appropriate than patterns indexed 0, 1, 2. The closer the relationship between the format of the data and the choice of patterns when compressing the better the smaller the counts may be written and the better the compression ratios. Considering that many executable binaries are in patterns of 4 bytes and information in databases is often stored in clusters of a fixed width, there is a strong advantage to viewing the file as data out of order.

4 Proof of Improvement Over Huffman

This proof is provided by Danny Sleator.

Note: For this proof only, I will redefine the way the file is indexed, so that $F(|F|)$ is the first character, $F(|F|-1)$ is the second character and so forth. $F(1)$ is the last character. This is important for the induction to run smoothly.

Definition 4.1 *H_n as the optimal Huffman tree for the characters $F(n), F(n-1), ..., F(1)$ in the given file F*

Definition 4.2 *$H_n(k)$ (requiring $k \leq n$) as the number of bits taken to represent the characters $F(k), F(k-1), ..., F(1)$ using the Huffman tree H_n.*

Definition 4.3 *$D(k)$ as the number of bits required to compress $F(n), F(n-1), ..., F(1)$ where you each $F(k)$ from 1 to n is compressed using H_k. This yields the formula: $D(n) = H_n(n) - H_n(n-1) + D(n-1)$*

Theorem 4.1 *Given an arbitrary file F, $D(|F|) \leq H_{|F|}(|F|)$*

Proof: By induction over n.

Inductive Hypothesis: assume $D(n) \leq H_n(n)$

Base Case: $D(1) = H_1(1)$ by definition of $D(1)$ therefore $D(1) \leq H_1(1)$

Inductive Case:

$D(n+1) = H_{n+1}(n+1) - H_{n+1}(n) + D(n)$

$H_{n+1}(n) \geq H_n(n)$ because H_n is optimal

$D(n+1) \leq H_{n+1}(n+1) - H_n(n) + D(n)$

$D(n) \leq H_n(n)$ by IH

$D(n+1) \leq H_{n+1}(n+1)$

∎

5 The Algorithms

For an algorithm to be in this class, it must store frequency counts to aid an adaptive statistical method (such as this decrementing Huffman) or to completely represent the file. The different algorithms in the class are defined by which counts are stored and in what order they are stored. Storing only the 0th order counts provides the decrementing Huffman algorithm I provided and proved bounds for above. More complicated lists of counts will produce smaller file sizes but larger headers. If the patterns in the data are matched closely to the chosen counts, the representation of those counts can be made small. The technicalities of how counts may be represented will be discussed in a later section. One thing to note when choosing which counts are useful in compressing a file is that some counts contain others. This was mentioned earlier in the section on deriving counts.

5.1 Using Counts to Compress a File

Given that we have stored some set of counts, we may use that information to compress the rest of the file. The number of bytes required to store additional counts is often much smaller than the number of bytes saved when representing the file given the additional data.

As I have mentioned before, storing the counts $(F, \{0\})$ provides that the file may be compressed to a size at least as small as the standard Huffman algorithm could provide. This, in most cases more than makes up for the size difference between representing the counts in the file header as opposed to just representing the Huffman tree in the file header.

Counts may also be used in higher orders to achieve greater compression. Say that the count $(F, \{0, 1\})$ is stored. We may construct $|A|$ decrementing Huffman trees, one for each $x \in A$, using the count $(F, \{0, 1\})_{\{(0,x)\}}$. Each tree represents the counts for the characters that

appear immediately after x in the file. This collection of Huffman trees contains first order data that usually produces significantly smaller compressed files. This ordering procedure may be extended into arbitrary dimensions with diminishing returns. Eventually, if the space of the counts is large enough, each Huffman tree has only 1 or 0 patterns in it, and the file has been completely represented by the counts.

5.2 Complete Representation of a File by Counts

As I just mentioned, if enough counts are stored, the file is reduced to nothing, and all that remains is the header containing counts. This header may actually be much smaller than the original file if the counts are stored in a cleaver way. The implementation I chose compresses the file in this way, and with good results.

The important thing to notice when storing multiple counts is when no more information is necessary. In most cases, when writing the next higher order count, almost all of its data may be derived from looking at lower order counts, and only a small amount more is required at each higher step. When no more information is needed to store the next count, the file has been completely represented and we may stop.

For every file, there is a minimum depth for which counts must be stored before that file may be completely represented by its counts. If the file is to be stored completely by its counts, they must be at least of this depth. For example, the text of Shakespeare's Hamlet, requires a depth of 80 before it is completely represented, even though 90

There is one technicality to clean up after the file is completely represented by counts. Because the file was defined to be cyclical (see definition of a file), the counts represent the file as a cycle and yield no way of telling which byte originally came first. Since there are $|L|$ possible rotations for this cycle, an additional $\log_2(|L|)$ bits are needed to denote which came first. This number is actually perfectly analogous to the index required by the BW transform.

6 Representing the counts

In order to make these algorithms practical, we must have a method of representing the counts in an efficient way. This turns out to be completely critical because in the end, we are representing the file completely as counts. The difficulty in representing counts stems from not knowing how many bits to use when representing each individual count.

My solution involving storing specifically chosen sums of a set of counts. First off, given two positive integers, call them x and y with $0 \leq x \leq y$ and y known, it only takes $\log_2(y+1)$ bits to represent x because there are only $y+1$ possible values for x. Using this logic, given

a set of numbers (assume we know their sum, call it N.

Note:this turns out to be a valid assumption when running this algorithm.) we may divide the numbers into two groups, as evenly as possible, and store the sum of one of those groups in $\log_2(N)$ bits. The sum of the other group may be obtained by subtracting the sum of the first group from N. Now we have two smaller groups, each with a known sum, and we may use the same procedure recursively until each group is of size only one. Since we know the sum of all the elements in a group of size one by the recursive procedure, in the end, we know the value of each number in the set.

6.1 The One Dimensional Case

In the one dimensional case it is easy to implement this procedure using an array.

I must make an aside quickly to require one more piece of information, an ordering on the alphabet A. The ordering may appear naturally, as with the bytes or letter, but may be arbitrary, as long as it is consistent throughout the compression and decompression procedures. Denote the ith character in A as $A[i]$. (The first character is $A[0]$ and the last $A[|A|-1]$.)

Define an array of non negative integers of length $2 \times |A|$, I'll call it Arr. For i from 0 to $|A|-1$ set $Arr[|A||+i] = (F, \{0\})(\{(0, A[i])\})$. For i from $|A|-1$ to 1 by -1, set $Arr[i] = Arr[2 \times i] + Arr[2 \times i + 1]$.

$Arr[1]$ turns out to be the sum of all of the counts, which we are assumed to know. Assuming we know $Arr[k]$ and $Arr[2 \times k]$, $Arr[2 \times k+1)$ may be obtained by taking $Arr[k] - Arr[2 \times k]$. Thus, inductively, we only need to store the even Array locations $Arr[2 \times k]$, we may do so each in $\log_2(Arr[k]+1)$ bits.

6.2 Example in the First Dimension

An example provides some motivation behind using this method to represent counts in one dimension.

Take $|F| = 15, A = \{a, b, c, d, e, f, g, h\}$ ordered in the natural way and $(F, \{0\}) = \{(0, a) \to 3, (0, b) \to 0, (0, c) \to 0, (0, d) \to 0, (0, e) \to 2, (0, f) \to 3, (0, g) \to 2, (0, h) \to 5\}$

$FirstTry$: Storing $(F, \{0\})$ using one machine word for each count, assumed to be 32 bits, the total space required to is $32 \times 8 = 256$ bits.

$SecondTry$: Given that we know $|F|$, and that $\forall x \in A(F, \{0\})(\{(0, x)\}) \leq |F|$, we may use only $\log_2(|F|+1) = 4$ bits for each count, 8 counts totals to 32 bits.

Note: We may now notice that storing 7 out of 8 of the counts is sufficient because the last one may be obtained by subtracting the sum of the 7 from the sum total, in this case $|F|$. This yields a 28 bits total.

$Using partial sums$ It will be beneficial to use the shorter notation for this example: $\{C\} := \sum_{x \in C}(F, \{0\})(\{(0, x)\}))$.

Write $\{a, b, c, d\}$ in $\log_2(|F| + 1)$ bits

Write $\{a, b\}$ in $\log_2(\{a, b, c, d\} + 1)$ bits

Write $\{a\}$ in $\log_2(\{a, b\} + 1)$ bits

Write $\{c\}$ in $\log_2(\{a, b, c, d\} - \{a, b\} + 1)$ bits

Write $\{e, f\}$ in $\log_2(|F| - \{a, b, c, d\} + 1)$ bits

Write $\{e\}$ in $\log_2(\{e, f\} + 1)$ bits

Write $\{g\}$ in $\log_2(|F| - \{a, b, c, d\} - \{e, f\} + 1)$ bits

This totals to 18 bits in this example. See **figure 1** for reference.

6.3 Higher Orders

In the one dimensional case with $|A|$ values to be stored, an array of size $2 \times |A|$ was be used to build a table partial sums. In the two dimensional case, where there $|A| \times |A|$ data to be stored, a matrix of size $(2 \times |A|) \times (2 \times |A|)$ may be used in a very similar way. Call this matrix M. For $x \in 0..|A| - 1, y \in 0..|A| - 1$ set $M[|A| + i][|A| + j] = (F, \{0, 1\})(\{(0, A[i]), (1, A[j])\})$. For the rest of the matrix, define: $M[x][y] = M[2 \times x][y] + M[2 \times x + 1][y] = M[x][2 \times y] + M[x][2 \times y + 1]$. Again only the values with even indices need to be stored in order to reconstruct the table. Higher orders may be stored analogously using cubes or hyper-cubes of partial sums. A naive implementation of this will take a huge amount of memory, because the table $(2 \times |A|)^{dimension}$ large. However it is very sparse and a hash table may be used effectively.

7 My Implementation

For the remainder of the paper I will provide an explanation of a working implementation of one of these algorithms, its performance relative to other algorithms in use, and some of its useful features. The algorithm I chose involves writing counts of higher and higher order until the file is fully represented by the counts. The chosen counts for this algorithm are $(F, \{\}), (F, \{1\}), ..., (F\{1, 2, ..., n\})$. The method of partial sums was only implemented

to the first order to store the counts because of time restraints and the difficulty of coding higher orders effectively. I will explain what I mean shortly.

The counts are stored in a data structure very similar to a trie where each pattern indexes its count. A diagram provides an intuitive example. See **figure 2**. This data structure provides for faster retrieval and sorting of counts than multiple hash tables do. Each level of the trie represents one collection of counts. The root node is $(F, \{\})$, the first level $(F, \{0\})$ and so forth.

7.1 Counting quickly

The first task was gathering all of the relative counts quickly and pruning out the counts that could be derived from a lower order. Counting is done by simply iterating through the file. Bucket sort is used to keep similar patterns close to each other in the file to aid speed when counting the higher dimensional counts.

The pruning condition goes as follows. Given pattern P, if for some $k \in 0..|P|-2$, $(F, \{1..|P|-1\})(P) = (F, \{1..k\})(\{(0, P(|P|-k)), .., (k, P(|P|-1))\})$ then counting pattern P is unnecessary. It is unnecessary because all of the characters that would appear after pattern P in the file will also appear after the shorter pattern. Each node also has a pointer back to its longest suffix, for speed purposes. These pointers do not show up in the graph in **figure 2**. However, notice in the figure , the stopping conditions holds for all of the *ed nodes. E.g. the pattern "ea" has the same count as the pattern "a", thus any character that appeared after "ea" in the file would also appear after "a".

7.2 The Resulting Data Structure

In the trie, any pattern that the stopping condition holds for simply don't exist as nodes (even though they show up *ed in **figure 2**. Any pointers that would point to them instead point the count with the shorter pattern. The resulting structure is **figure 3**.

Given any pattern in a file, we may produce a new pattern of greater length also in that file by finding an occurrence of the pattern and appending the byte that appears immediately after it (in the file). Thus each node in our trie will have at least one child. This would seem to lead to an infinite data structure, but because of the end condition some children loop back we get a cyclical data structure. See **figure 3**. Notice that since each node points to at least one other, and there are a finite number of nodes, there must be a cycle. This cycle is the represented file.

The data structure constructed contains the file as a cycle of nodes. However, since this cycle of nodes doesn't have a beginning or end, we need an additional $\log_2(|F|+1)$ bits to where in the cycle is the break between beginning and end. Calculating this index, and

following it is not difficult. I will not discuss it here.

The file can be reconstructed from the data structure simply by finding the beginning and then reading the characters off the cycle.

Compression is achieved by the fact that it is often much smaller to represent this data structure than it was to represent the original file. The method of compression is explained in the next section.

7.3 Simplified Partial Sums

Since higher order partial sums turned out to be very difficult to code effectively, for this implementation, I chose to compress each node's children as a separate count and used the only first order partial sums method. The first order partial sums method on its own works fine, but works much better if you realize that the collection of counts at pattern is a subset of the collection of counts at any pattern of shorter length contained as a suffix of P. This provides tighter bounds for what each count can be and reduces the number of bits it takes to represent them. This is the largest simplifying assumption I made when writing this program.

7.4 The Results

The results of representing a file as a cyclical trie of counts in this way and storing the counts using partial sums are surprisingly good. On average, the file size produced is comparable to those produced by the zip standard. The zip results produced in **chart 1** are by WinZip set to maximal (slowest). It still falls short of the 12 Ghosts implementation the Burrows Wheeler transform with MTF encoding. (Notice that the 12 Ghosts program produces self extracting executables, which contain 38K of executable code in addition to the file size.) Considering the gross simplifying assumption of representing the counts with only first order partial sums, these results are very promising.

Along with the compression, there are many other benefits of storing a file in this cyclical trie structure. Most of them are based the fact that the file is easily searchable in this form.

Since our structure is a trie, searching for a pattern within the file or determining the number of times a pattern occurs within the file is very fast and in general the number of steps required for the search is on the order of $|P|$

Because we can search for patterns in the file quickly, and DFS does so in lexicographical order, we may produce a Burrows Wheeler transform in linear time by simply running DFS on this data structure. Because all patterns will be output in order this is equivalent to sorting all rotations of the file.

Also due to the fast searching capability, we may use this structure as a compressed dictionary that resembles a normal trie. Using special characters to mark the beginning of keys and values, we may search quickly for a key and check if it is followed by a value, if it is, we may read that value. The whole structure remains in the compressible trie form. **figure 4** shows a compressed dictionary structure.

The compressed dictionary idea leads to a host of other possibilities, including the idea of a compressed, easily searchable file system. Pathnames may be used as keys and files as values. A file system of this type would also maintain the benefit that each file it stores would be easy to search. Finally, since all of the files would be contained in one archive, the patterns could build off of each other, and the total archive size could be much smaller than if each file had been compressed individually.

8 Other Uses/Applications

Video and Image compression: Depending on how much is known about the files to be compressed, more specific algorithms may be applied for better results. I mentioned earlier the idea that counts do not have to be linear, and can be spaced to fit patterns in the data such as bitmaps or databases. With bitmaps, patterns may be chosen such that not only do patterns look from one pixel to the one after it, they may look from one pixel to the one below it. Such a pattern may be $\{0, 3, 3 \times (imagewidth)\}$. Patterns like this allow the file be viewed as a 2 dimensional matrix as opposed to simply an array of bytes. When compressing video, patterns could also include the pixel from the previous frame thus viewing the file as a three dimensional block of data, or a one dimensional array of two dimensional objects.

Frequency counts may also be used to implement lossy compression algorithms. This may be done by reducing the alphabet size in any way. When compressing say, an image, if the low order bit of each byte was dropped, the alphabet will be effectively reduced in half, and quite a bit less data will need to be stored. More optimally, the size of the alphabet for each count may be adjusted dynamicly as function of data already known by both the compressor/decompressor.

Machine Learning: The last nice application I will mention of frequency counting methods and statistical methods in general is that frequency counts make good features for any machine learning technique. Machine learning can be used with counting methods to determine the type of a file in order to apply more specific algorithms to that file. Machine learning may also be used to classify files in many other ways, such as "Is this an ASCII file?" or the ever famous "Is this piece of mail spam?"

9 Conclusion

I have presented a generic class of lossless data compression algorithms based on counting that provide useful features as well as excellent compression ratios. I have also provided a working example of one such of these algorithms and some results of its performance.

I believe that this class of algorithms will be useful in the future because of the beneficial features it provides in addition to competitive compression ratios. Being able to search a file in a fast manner without completely uncompressing it may be invaluable.

As was the initial goal, the compression achieved by these algorithms at least rivals LZW and the zip standard. With more work, perhaps they will surpass even Burrows Wheeler some day.

10 Notes on Other Algorithms

10.1 Huffman

Huffman compression is a simple information theoretic algorithm that uses a tree structure to decide the optimal number of bits that each character in an alphabet should use when representing a file. Characters that occur more frequently in the file use fewer bits than characters that occur more rarely. The resulting file after recoding each character according to the Huffman tree is guaranteed to be at least as small as the original file, and is optimal for the information given. See Data Structures & Problem solving [3].

10.2 LZW

LZW is a pattern building algorithm. It is also the most common algorithm in use today and is the basis for the zip standard. The algorithm works effectively as follows.

Start with the standard alphabet of 256 characters, each with its own 8 bit representation. The alphabet grows throughout the procedure. Each iteration adds on pattern to the alphabet. The number of bits taken to represent one character in the alphabet is always increasing, but each character is held to a restriction that its representation is no more than one bit longer or shorter than any other the representation of any other character.

At each step, we read the longest pattern at the current file offset that we have in our alphabet. Then we write the representation of that character (pattern) and add a new pattern which consists of the written pattern and the next byte of the file to the library. See Data Structures & Problem Solving [3].

10.3 Burrows-Wheeler (BW)

The Burrows Wheeler transform is an algorithm that performs a reversible sorting procedure to the file. It does so in a way that no data is lost, but similar characters are collected in similar locations.

Once the file has been mostly sorted, a move to front encoding is applied to the file. MTF encoding is a procedure where at each step in the file, characters that have occurred recently in the past take fewer bits to represent. Since the file is mostly sorted, MTF works well on files that have been sorted using BW. In practice, the BW algorithm works better than most other algorithms. See Block-sorting Lossless Data Compression Algorithm [1].

11 Explanation of Figures

figure 1 demonstrates the method of partial sums in the first order. The *ed counts are written. The array is an implementation of the tree.

figure 2 is trie representation of the file ”she sells sea shells by the sea shore” with the stopping condition emphasized

figure 3 is figure 2 where each child pointer is followed correctly. Notice the loop in the graph.

figure 4 is an example dictionary structure, containing two key-value pairs, (frodo, elijah wood) and (gandalf, ian mckellen), # denotes the beginning of a key and @ the beginning of a value.

chart 1 is a graph of results when compressing a set files using 4 different algorithms.

12 References

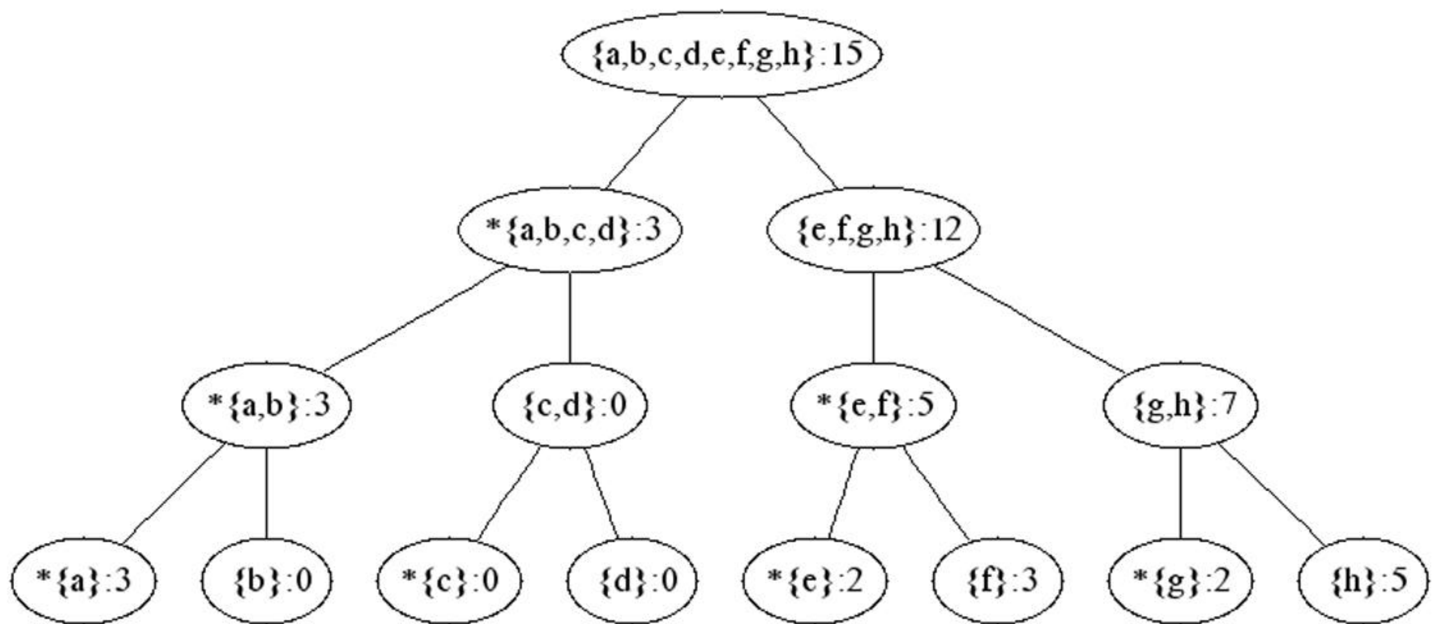
[1] M Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. SRC Research Report. No. 124. May 1994.

[2] R. Grossi, A. Gupta, and J.S. Vitter. High-Order Entropy-Compressed Text Indexes. In Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA’03)

[3] M. Weiss. Data Structures & Problem Solving using JAVA – 2nd ed. Addison Wesley 2002.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms – 2nd ed. The MIT Press. Cambridge Massachusetts. 2002.

figure 1



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	15	3	12	3	0	5	7	3	0	0	0	2	3	2	7

figure 2

(-):37

(_):7 (a):2 (b):1 (e):7 (h):4 (l):4 (o):1 (r):1 (s):8 (t):1 (y):1

_(b):1* _(s):5 _(t):1* a(_):2 b(y):1* e(_):2 e(a):2* e(l):2 e(s):1 h(e):3 h(o):1* l(l):2 l(s):2 o(r):1* r(e):1 s(_):2 s(e):3 s(h):3 t(h):1 y(_):1

_s(e):3* _s(h):2 a_(s):2 e_(s):2 el(l):2* es(h):1 he(_):2* he(l):1 ll(s):2* ls(_):2* re(s):1* s_(b):1* s_(s):1 se(a):2* se(l):1 sh(e):2 sh(o):1* th(e):1 y_(t):1*

_sh(e):1 _sh(o):1* a_s(h):2* e_s(e):2 esh(e):1 hel(l):1 s_s(e):1 sel(l):1 she(_):1 she(l):1* the(_):1

_shel(l):1* e_se(a):1 e_se(l):1* eshe(_):1* hell(s):1 s_se(a):1 sell(s):1 she_(s):1 the_(s):1

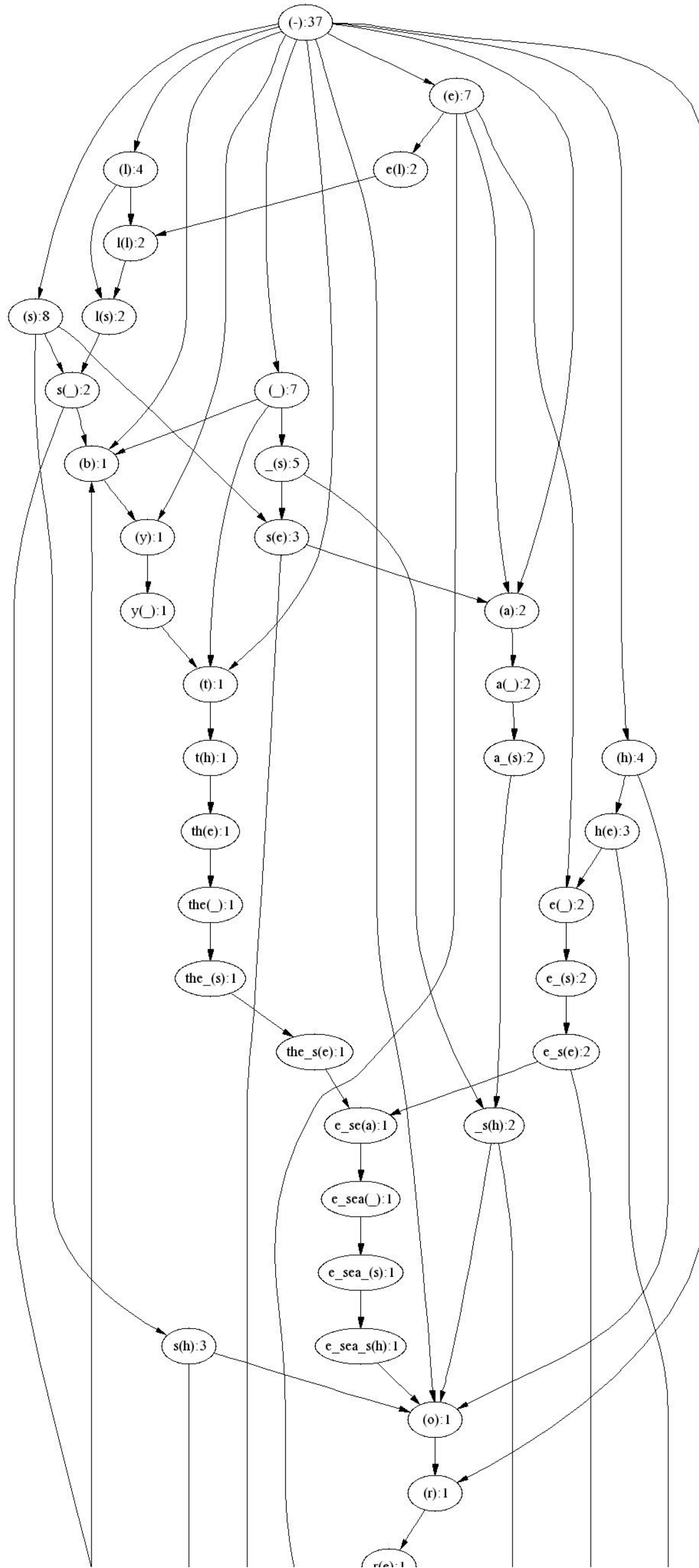
e_sea(_):1 hells(_):1 s_sea(_):1 sells(_):1 she_s(e):1 the_s(e):1

e_sea_(s):1 hells_(b):1* s_sea_(s):1 sells_(s):1* she_se(l):1* the_se(a):1*

e_sea_s(h):1 s_sea_s(h):1

e_sea_sh(o):1* s_sea_sh(e):1*

figure 3



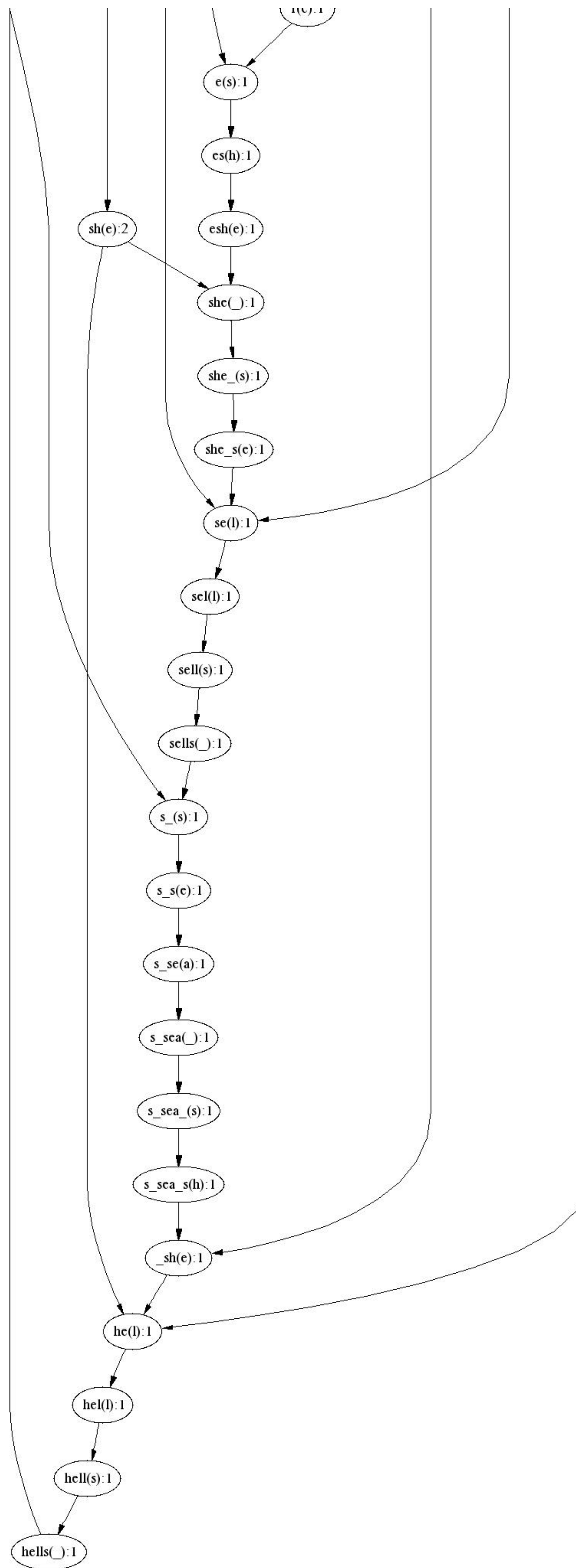
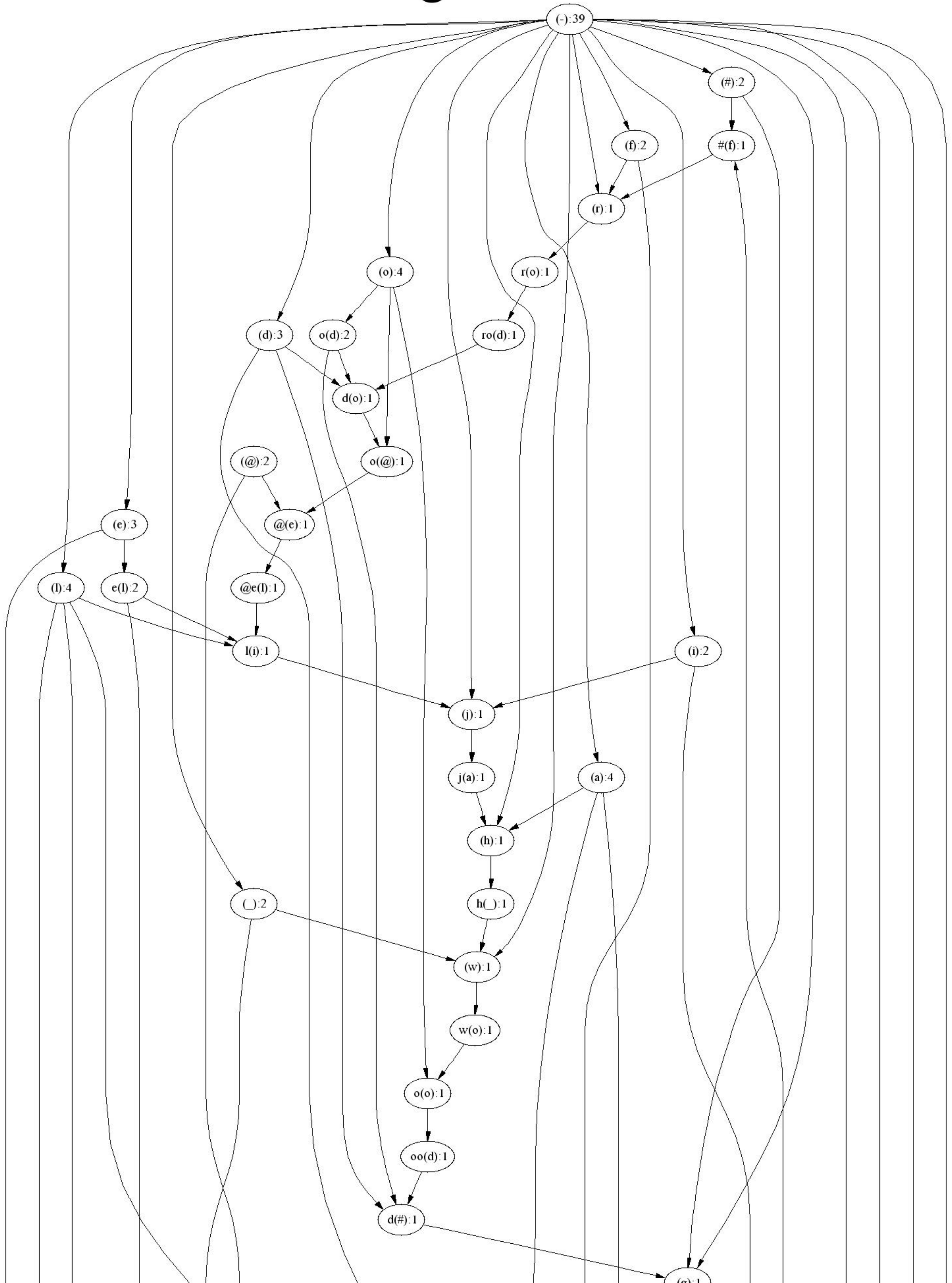


figure 3 (continued)

figure 4



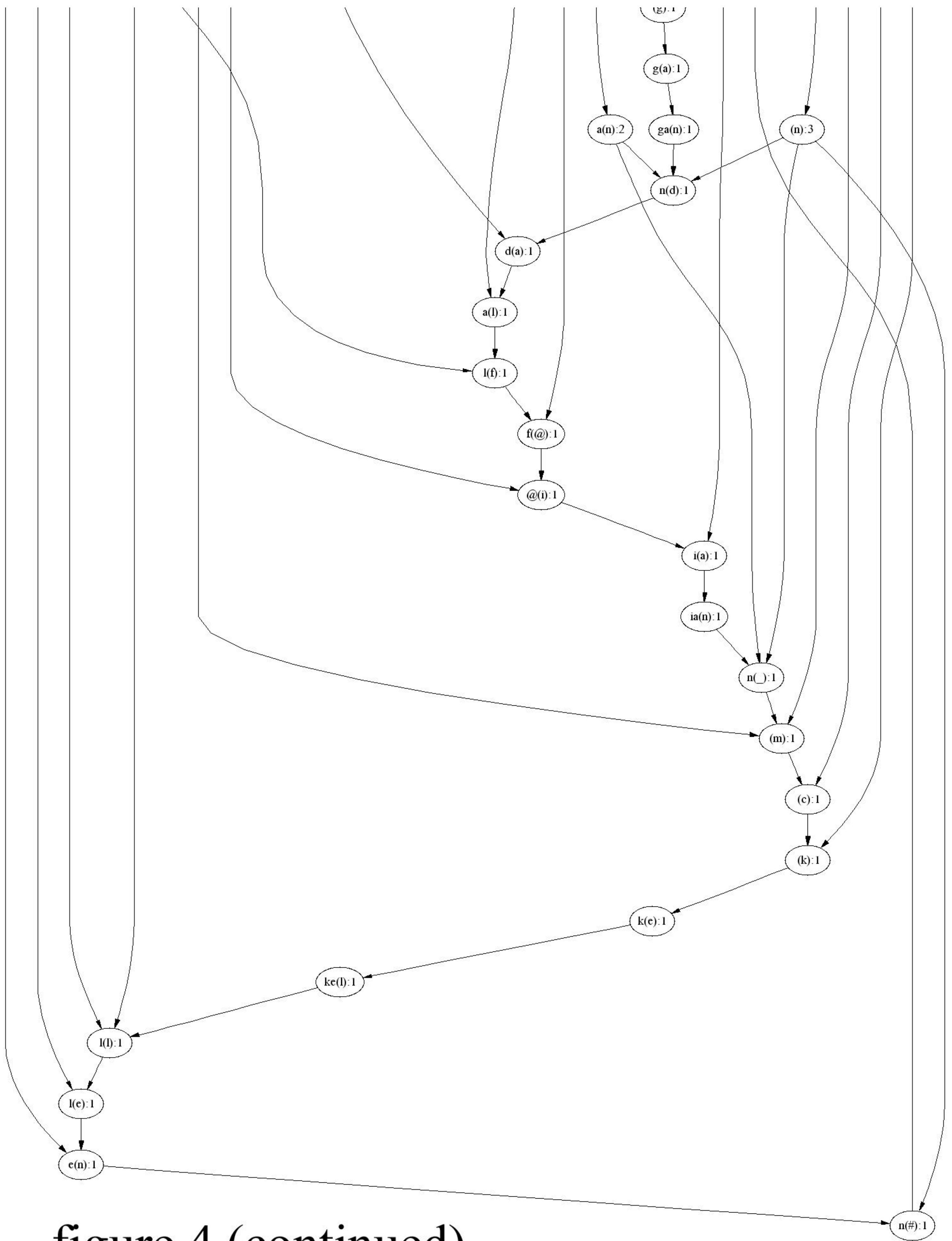


figure 4 (continued)

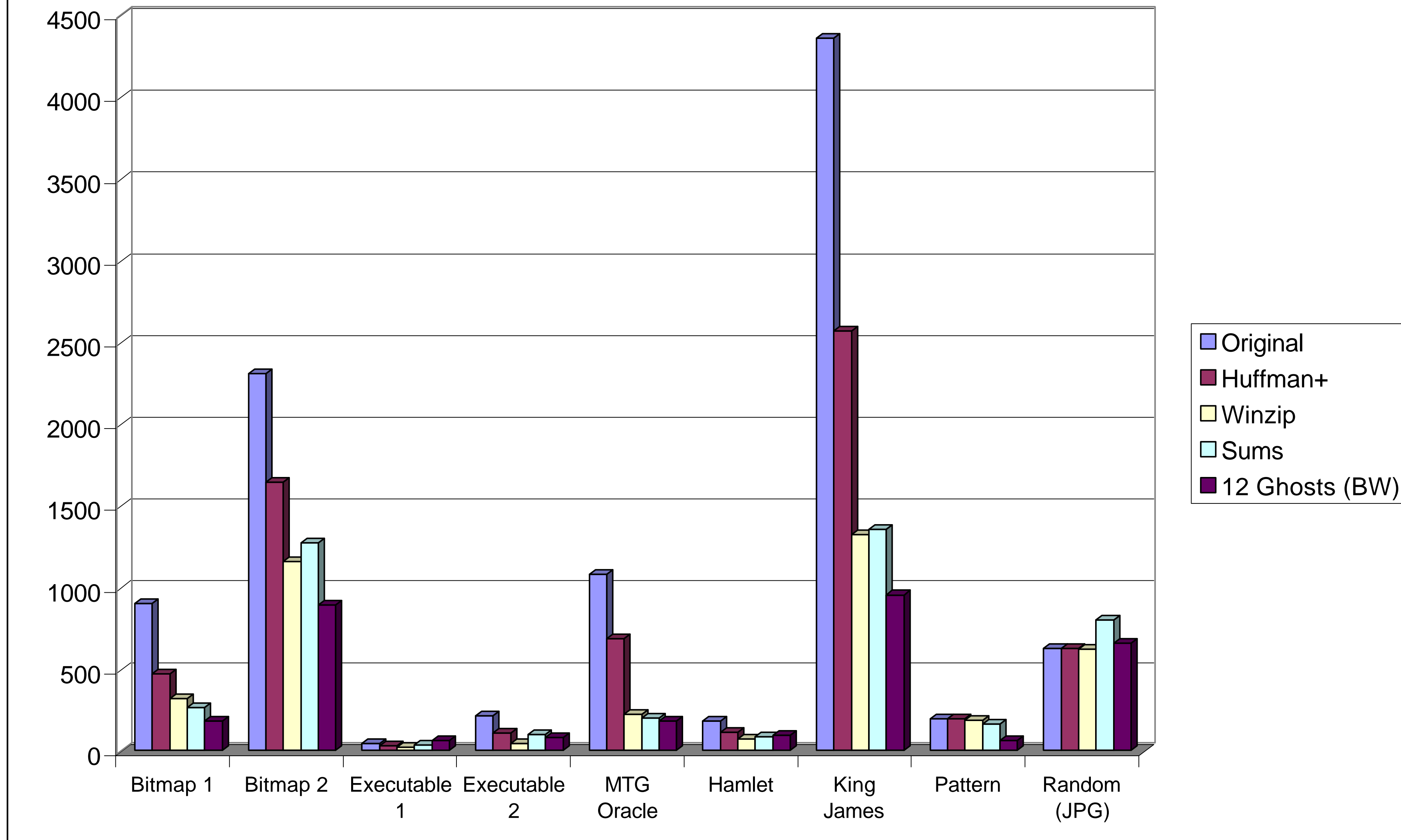


chart 1
4500
4000
3500
3000
2500
2000
1500
1000
500
0
Bitmap 1
Bitmap 2
Executable 1
Executable 2
MTG Oracle
Hamlet
King James
Pattern
Random (JPG)
Original
Huffman+
Winzip
Sums
12 Ghosts (BW)