Acta
Technica
Jaurinensis

Vol. 10, No.1, pp. 1-12, 2017
DOI: 10.14513/actatechjaur.v10.n1.405
Available online at acta.sze.hu

On Generalizations and Improvements to the Shannon-Fano Code

D. Várkonyi[1], P. Hudoba[2]

**[1]Eötvös Loránd University, Faculty of Informatics, Department of Numerical Analysis
Pázmány Péter sétány 1/C, 1117 Budapest, Hungary
Phone: +36 1 372 2500
e-mail: varkonyidavid91@gmail.com**

**[2] Eötvös Loránd University, Faculty of Informatics, Department of Computer Algebra
Pázmány Péter sétány 1/C, 1117 Budapest, Hungary
Phone: +36 1 372 2500
e-mail: hudi1989@gmail.com**

Abstract: This paper examines the possibility of generalizing the Shannon-Fano code for cases where the output alphabet has more then *2* (*n*) symbols. This type of generalization is well-known for the famous Huffman code. Furthermore, we will be looking at possible improvements to the algorithm, as well as other entropy based lossless data compression techniques, based on the same ideas as the Shannon-Fano code. All algorithms discussed in the paper were implemented by us in C++, and we will be illustrating our hypotheses with test results, made on an average performance PC.

Keywords: compression, lossless, entropy, Shannon-Fano

1. Introduction

Shannon-Fano coding [1][2] is an entropy based lossless data compression technique, which means that it is an algorithm for constructing a prefix code, in a way that a message coded in this will be shorter than normally. First, we need to define what we mean by "normally".

Most coding schemes are developed explicitly for binary encoding, since modern computers use the binary system, however we will be encoding into an arbitrary system.

Let

$$l: \Omega \rightarrow L, |L| < +\infty \tag{1}$$

be a random variable, and the possible outcomes of l (the elements of the alphabet) letters. We will be working with finite alphabets, as shown above. Next we will be sampling this random variable n times, giving us a "message" of length n. This gives us:

$$l_1, \dots, l_n \colon \Omega \to L \tag{2}$$

This is a sequence of independent and identically distributed (IID) random variables. We will only be working with independent variables, however finding encoding schemes in cases where the letters are not independent of each other is of great interest, and it is a subject of dictionary based compression methods. These methods try to eliminate the redundancy of information in saying that the *i*th letter is *x*, when the *i*th letter is almost entirely (or entirely) determined by the previous (or following) letters. However this is not our case, we will be working with independent variables, meaning the *i*th letter is not at all determined by the other letters.

The concatenation of these letters will give us the message. So we arrived at:

$$m \colon \Omega \to M = L^n, m = l_1 \dots l_n \tag{3}$$

Now, we will be encoding our message in a “different alphabet” (which we will call symbol set) of smaller size then “originally”. Let our symbol set be:

$$|A| < |L| \tag{4}$$

The elements of the *A* will be called symbols. Since (4), we need to encode each letter with a sequence of symbols. The trivial solution would be to encode each letter in exactly

$$\lceil \log_{|A|} |L| \rceil \tag{5}$$

symbols. This way assigning symbol sequences to letters would be simple. Let

$$\tilde{l}_1, \dots, \tilde{l}_{|L|} \; and \; \tilde{a}_1, \dots, \tilde{a}_{|A|} \tag{6}$$

be an arbitrary ordering of sets *L* and *A* (all elements of the set occur exactly once in the sequence). Our “simple” assignment would then be:

$$f \colon L \to A^{\lceil \log_{|A|} |L| \rceil}, f(\tilde{l}_i) := \tilde{a}_{\left\lceil \frac{i}{|L|/|A|} \right\rceil} \tilde{a}_{\left\lceil \frac{i - \left(\left\lceil \frac{i}{|L|/|A|} \right\rceil \frac{|L|}{|A|} \right)}{|L|/|A|} \right\rceil} \dots \tilde{a}_{\left\lceil \frac{i-(\dots)}{|L|/|A|} \right\rceil} \tag{7}$$

With this conversion between our two alphabets, our message encoded becomes:

$$m_e \colon \Omega \to M_e = A^{\lceil \log_{|A|} |L| \rceil n}, m_e = f(l_1) \dots f(l_n) \tag{8}$$

This is the trivial encoding, and thus by the “original length” of the message we will mean the number of symbols we can encode it in, using this technique, meaning the number of symbols needed to encode *1* letter, times the number of letters, so:

$$\lceil \log_{|A|} |L| \rceil n \tag{9}$$

Let us look at an example. Let our symbol set be binary, and let it consist of 0 and 1.

$$|A| = 2, \tilde{a}_1 = 0, \tilde{a}_2 = 1 \tag{10}$$

This is the case when we want to encode data on binary computer. We will encode a letter in as many symbols as we need, in order to be able to differentiate the letters (no two letters can have the same symbol sequence assigned to them). Naturally, we need

$$\lceil \log_2 |L| \rceil \tag{11}$$

symbols. Then we set the first symbol to 0 for each letter in the first half of the alphabet, and to 1 for the second half. We do this recursively on both halves, and we arrive at a binary encoding of our message.

This works great when the size of our alphabet is a power of $|A|$ (*2* in the binary case), and the letters have a uniform distribution, meaning that the *i*th letter in the message has the same probability being a certain element from the alphabet than any other. If either of these two conditions are not met, redundancy appears in the encoding.

If the size of the alphabet is not a power of $|A|$, symbol sequences will remain with no letters assigned to them. This is a “waste” of symbol sequences, a redundancy in the code.

If the letters’ distribution is not uniform, then *i*th letter is “partially determined” by its distribution alone, and thus stating it is x is “partially redundant”.

Both of these redundancies can be eliminated by clever assignment of symbol sequences to letters (*f*).

The only thing we need to keep in mind is that not only f has to be invertible (which is trivial), but the entire encoding scheme:

$$E: (\Omega \to M) \to (\Omega \to M_e), E(m) = m_e \tag{12}$$

Encoding concatenates the values of f, so if f is invertible but has in its range a symbol sequence which is a prefix (initial segment) of another symbol sequence, also in its range, the encoding can still be uninvertible. For example, let:

$$f(a) = 0, f(b) = 010, f(c) = 10 \tag{13}$$

With this choice, the encoded message *010* could have been encoded from *ac* or *b*.

To eliminate this possibility, and finding invertible encodings, we will only be constructing prefix codes (no symbol sequence in range of f can be a prefix of any other symbol sequence in range of f).

In most cases we will be given a message which we need to compress, which means that instead of knowing the actual distribution of l, we will have an empirical distribution.

2. The Shannon-Fano code

As stated previously the Shannon-Fano code is an algorithm for constructing prefix codes for data compression. It was developed for binary output ($|A|=2$). The algorithm follows 3 steps:

1. Order the set of possible letters into a list in descending order of their probability of occurrence (or number of occurrences). Place the list in the top node of a binary tree graph.
2. Cut the list into two sub-lists, in a way that the sum of the probabilities (or occurrences) in the two sub-lists are as close to each other as possible. Place

each sub-list in a child node of the list. Repeat this step recursively until no leaf node remains with more than *1* letters.

3. Assign bit sequences to the letters based on the binary tree, in the same way as Huffman's code [3]: for a given letter, walk down the tree to its node, getting a *0* from each left branch taken, and a *1* from each right branch.

The point of this algorithm is to maximize the entropy of each bit in the output one by one. Let us examine this statement a bit closer.

Let self-information of an event be the "surprise" this event means, the improbability of the event occurring:

$$I(l = 'x') = \log\left(\frac{1}{P(l='x')}\right) \tag{14}$$

Let the entropy of a random variable be the expected value of self-information the outcomes of the random variable could mean.

$$H(l) = E(I(l)) \tag{15}$$

This quantity will be maximal if the distribution of *l* is uniform.

Now, for a given symbol sequence which encodes a letter, the first bit determines whether the letter is on the left or the right branch of the root node. That first bit has maximum entropy if the distribution of its outcomes is uniform, or as close to uniform as possible. This is exactly what the Shannon-Fano code achieves by cutting the list in a way that the sum of probabilities (or occurrences) are as close to each other as possible.

After the first bit eliminated the maximum amount of uncertainty it possible could, the list cutting is called recursively, which maximizes the entropy of the second bit, and so on.

Altogether this maximizes the entropy of each bit, one by one.

Intuitively this algorithm could lead to an optimal code, but that is not the case. The problem is that while it achieves optimality bit by bit, it fails on a global scope.

If cutting a list at a certain place is optimal, it could be that after this cut we face much worse choices in the next step (in cutting the sub-lists) as if we had cut at another place, and overall we get a worse code then we could have, using a suboptimal cut at the first step. Basically the algorithm steers the code in the way that seems optimal in a local context, but fails to recognize the global optimum. In this way Shannon-Fano code is a greedy alternative to Huffman's code.

3. Generalization for *n>2*

Generalization of this type of compression to not only binary output alphabets could be of great interest, for example because since a lot of modern telecommunication methods use a *>2* symbol rate, which means a direct compression of the transmitted message could yield greater bandwidths.

Huffman's algorithm, while mostly discussed in the binary case, has an easy and well-known generalization for the *n>2* case. Huffman himself considered this case in his original paper in 1952. However, making this generalization for the Shannon-Fano code is not as straightforward.

In Huffman's algorithm, in binary case the algorithm needs to find the *2* lowest probability (or occurrence) node. For a general n, we simply find the n lowest probability (or occurrence) node, and the algorithm works the same as before.

Shannon-Fano code cuts a list into *2* pieces in a way that minimizes the difference of the sums of probabilities (or occurrences) between them. For a general n, we would need to cut the list into n pieces in such a way. This maximizes the entropy of the next symbol in the output, same as before. The problem that for $n>2$ there is no such thing as the "difference of the sums of probabilities (or occurrences) between them", there only exists differences of the sums of probabilities (or occurrences) between them.

We need to redefine what we mean by optimal cut. The underlying idea will remain the same, meaning that we want to set the distribution of the next symbol to as close to uniform as possible (and thus maximizing its entropy), for which we need a way to measure how "bad" a distribution is, how "far" it is from uniform.

Definition of this quantity is not trivial. We could use any number of such measures, however intuitively, two seems obvious:

- Let the maximum difference of the sums be minimal.
- Let the sum of the differences of the sums be minimal.

If we think of the differences between the sums as a collection of numbers (a vector) these measures correspond to the p-norms of $p = \infty$ and $p = 1$, the two extrema, thus it is likely that one of these methods will yield the best results.

We examined both of these methods, and tested them extensively. The results are not conclusive, but tend to favour the second method ($p = 1$). Before looking at the test results let us discuss the other problem in generalizing the Shannon-Fano code.

The code cuts a list into n pieces in each step. We need to think about what happens when a list has less than n letters. In the binary case, this could not occur, because the only way a node would have less than *2* letters, is if it has *1*, which is the stop condition of the recursive cutting, so it is not a problem. In the general case however this is very much a possibility. There are also two solutions to this problem that seems obvious:

- Let us leave empty nodes: If we have $m<n$ letters on a list, cut the letters into a separate sub-list each, and leave n-m lists empty. This means that symbol sequences would remain unassigned, which is clearly a "waste of symbol" in a way.
- Let us cut the list in a way that rules out that down the tree this problem could occur. To guarantee this we need to only cut the list in a way that creates sub-lists that have m letters on them, where

$$(m-1) \equiv 0 \ (mod\ n-1) \tag{16}$$

We will prove that this constraint guarantees that a full subtree can be created from the node in a constructive manner:

Clearly, if *m=1*, then a full subtree can be created, since the node itself will be the tree. *m=1* also satisfies the (16) equation. If we want to add another letter to the node however, we need to add not only one, but *n-1*, because adding only *1* would create empty nodes, which are filled in by adding a total of *n-1* letters. By induction we can conclude that a full tree can be created by adding a multiple of *n-1* letters, so (16) must be true.

With this method however we may not be able to choose the optimal cut, which results in suboptimal symbol distribution. The other problem that arises when choosing this method is that the first node in the tree might not be such, that a "full" tree can be created, meaning that there are inputs for which this method cannot be applied.

We also examined both two of these options, and tested extensively. The results are fairly conclusive, and favour the second method, where we avoid leaving empty nodes.

We tested our algorithms with random text generators, and books available on the internet as well. Table 1 shows a typical result we got, while testing on a 100 paragraph text generated by an online random text generator. The empty rows are cases where the algorithm can not be applied.

Table 1. Test results on a randomly generated 100 paragraph text

TEXT LENGTH	BINARY	3-ARY	4-ARY	5-ARY
ORIGINAL	364998	243332	182499	182499
HUFFMAN	260087	166335	132717	115893
S-F MAX	260559	172763	148761	132904
S-F SUM	260559	172763	148565	132904
S-F MAX S	260559	166335		119506
S-F SUM S	260559	166335		115893

The left column labels the different methods we tested. The original and Huffman rows show the original length of the text, and the length using the Huffman coding. The "S-F MAX" rows show the results we got while using the $p = \infty$ norm, and the "S-F SUM" rows show the $p = 1$ norm, and the rows labelled with an "S" at the and show the results while avoiding creating empty nodes, and the unlabelled rows show the results we got while creating empty nodes, when needed.

The test results we got (not only those shown above) clearly show that the "S-F sum S" method performs the best of this 4, almost reaching Huffman's code.

4. Non-ordering Shannon-Fano

We will now examine what happens when we skip the ordering step from the algorithm. Ordering the set into a list can keep us from finding the optimal cut. Thinking of a set of m numbers it is entirely possible, that the optimal cut (measured by either one of our methods) would create subsets that are not strictly increasing. For *n=2* examine the case when we have numbers $\{1, 5, 6, 10\}$. Clearly the optimal cut is $\{1, 10\}$ and $\{5, 6\}$, however we could not have reached this if we pre-ordered the numbers into a list, and only cut them accordingly, because this way we would have created lists $\{1, 5\}$ and $\{6, 10\}$. Figure 1 shows an illustration of the phenomenon, on the text "abbbbbccccccdddddddddd", and *n=2*.

The ordering step's main role is to make the cuts faster. Still thinking of the *n*=2 case, if a list is ordered, the optimal separation can be found by moving a separator from left to right between the letters, and stopping when the right side becomes smaller (in sum) then the left one. Not only so, but this advantage is "inherited" down the tree. The list cut in such fashion becomes two ordered sub-lists, for which finding the separation can also be done in this manner.

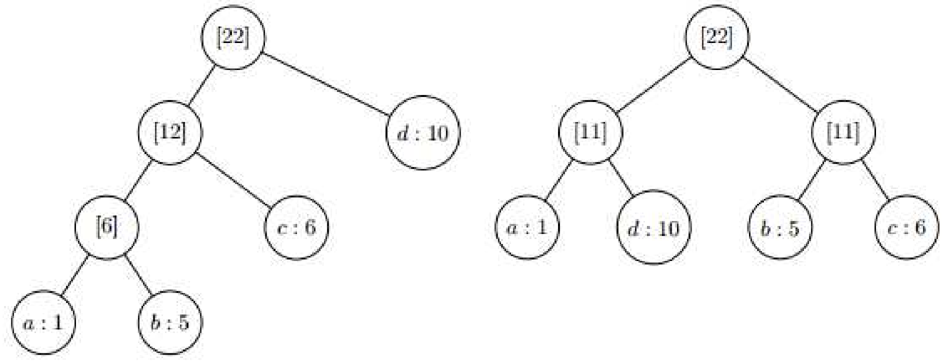

Figure 1. Ordering vs. Non-ordering algorithms

If we skip ordering, "finding the optimal cut" means that we need to cut a set into a disjoint union of *n* subsets, for which a certain measure of entropy is maximal.

Even in a simple case, for *n=2*, this is equivalent to the subset sum problem, which is NP-complete. In this case we are looking to find a subset of a set that sums up to as close to half the original set's total value as possible. Clearly, a simpler problem would be to find a subset of a set that sums up to exactly the original set's total value, which is a given number. This is the subset sum problem.

Overall, abandoning the ordering step will slow down the algorithm considerably, but could possibly improve the compression ratio.

In practice though, another factor comes into play. Even though ordering the set into a list can keep us from obtaining the locally optimal cut, it does not actually say anything about the globally optimal cut. It is possible that the non-ordered Shannon-Fano code, while getting better cuts locally, has a worse result globally. In fact, if we have a distribution which has a few very large numbers, and a lot of small ones (which means that the optimal compression ratio is high), the ordered Shannon-Fano keeps the small number of big numbers together, and creates a small subtree for them, and keeps a lot of small numbers together, creating a big, deep subtree for them. This guarantees the big numbers will be high on the tree, and the small ones will be on the bottom, getting us a good code. In the orderless version however, if the algorithm sticks one small number to a big one, that small number forces the big one down the tree another level, which is a great waste of information. So intuitively, it is entirely possible that the orderless algorithm gets worse results.

Figure 1 illustrates such a case. In this case the total output length using the ordered algorithm was 40, while using the non-ordered it was 44.

We implemented all our S-F variants in a non-ordering version, and tested them extensively. The tests conclude that not only the non-ordered algorithms run so slow, that it is practically impossible to apply them in any real-life situation, they result in a worse code (on a global scope) then their ordered counterparts for almost all inputs. These algorithms thus are clearly not worth investigating them further.

5. Distributing method

Up to this point we were constructing variants of the Shannon-Fano method by altering the way the algorithm chooses where to cut the set or list of the letters. We will now look at a completely different method of dividing a set of letters into *n* sets.

Let us create *n* empty sets, and assign our letters to one of these sets one-by-one. Take the letter with the highest probability (or most occurrences) and assign it to the set that has the least probability (or occurrence) in sum. Repeat this step until all of our letters have been assigned to a set. We are thus distributing the letters between the sets, in a way that keeps the distribution fairly uniform.

This method however has a defect. In a given step, if all but one set (the ith) have been assigned $1, n, n^2, \ldots$ letters (the sets have a power of n number of letters), then it is almost surely better to assign the next letter to the ith set, regardless of the values on the nodes. The reasoning behind this, is that on the next level down the tree, the "full" nodes will leave no empty nodes after their division. A node with 1 letter becomes a leaf, a node with n letters becomes a 2 level full tree, a node with n^2 letters becomes a 3 level tree, and so on. Adding any letter to a full node would trigger building another level somewhere in their subtree, which is clearly a waste, because it lowers other letters in the full tree, giving them a longer symbol sequence. If we assign the next letter to the ith set however, no other letters have to be forced down a level, and we get a better code.

There are exceptions to this argument. If the empty node is so far down the tree, that assigning the next letter there would result in such a long symbol sequence, that it is better to lower some letters higher up in the tree, and assign the next letter to the empty node we created, then this argument would be invalid.

This however is a very unlikely scenario, and happens only in extreme cases. In most cases, it is worth it to always fill the sets to "full" first, and then choose one to spoil. It is also possible to correct this flaw in the algorithm, by only assigning a letter to a non-full node, if down the tree it will not give us worse results overall then we would get if we spoil another full node. This however would require that we know information about the full tree in advance. It is possible to "try out" the assignment to the non-empty set, and calculate in advance if it would be a better choice, but that would be less of a Shannon-Fano code, as it would use information on the full tree to make a decision, and not only information on the level in question. It would no longer be an algorithm which builds a tree from top to bottom, by splitting its branches recursively, which is in essence what the Shannon-Fano code does.

Figure 2 illustrates a case where dividing into sets that have a power of *n* letters clearly shows its advantage. Our text is "aaaaaabbbbbccccdddee", and *n=2*. On the left side, the optimal division of the first list caused empty nodes to be created on the second one, while on the right side no empty node is created. Using the first method we would need 34 symbols to encode our text, while using the second one this number is only 29.

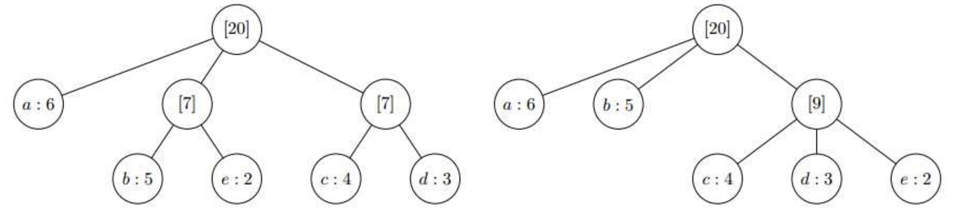

Figure 2. Distributing methods

We implemented both versions of the distributing method, an algorithm where we do not care about empty nodes (labelled "S-F DISTRIBUTING"), and one where we fill to "full" first (labelled "S-F DISTRIBUTING S"). Both algorithms have been tested extensively, and overall we found that the latter one performs better in almost all cases. Table 2 shows our test results, on the same input text as the previous tables.

Table 2. Distributing method

TEXT LENGTH	BINARY	3-ARY	4-ARY	5-ARY
ORIGINAL	364998	243332	182499	182499
HUFFMAN	260087	166335	132717	115893
S-F DISTRIBUTING	268187	179727	141954	125097
S-F DISTRIBUTING S	268187	171117	132817	117311

6. Pushdown method

As stated previously, the essence of the Shannon-Fano coding technique is to build up a tree by starting from a root node, and recursively splitting it into branches. For our last method we will be stepping outside this framework somewhat.

The distributing method assigns each letter to a node in the next level one by one, and after it is finished, it is repeated on all nodes created in the next level, recursively. Let us construct a similar method, one which assigns letters one by one, but assigns them to a node anywhere in the tree, not only the next level. This way no recursive calling will be required, as we place the letter in the tree, not only decide on which branch will it reside.

This is no longer a Shannon-Fano method in the way Shannon-Fano methods work level by level. It is however still a Shannon-Fano method in the way that it will make locally optimal choices.

The only thing we need to work out is how we assign a given letter (*l*) to a node in the tree. We can:

- Place it on an empty node on the *i*th level
- Divide a node on level *j* in the tree which has letter *L* assigned to it into another level of *n* nodes, assign *L* to one, *l* to another, and thus create *n-2* empty nodes.

Let us consider how many extra symbols would appear in our output if we would do these steps:

- If we place it on an empty node on the *i*th level, letter *l* would be assigned a symbol sequence of length *i*. If letter *l* has *P(l)* occurrences that would mean *i*P(l)* extra symbols.
- If we divide an existing node, that means that *l* will be on the *(j+1)*th level, which gets is *(j+1)*P(l)* extra symbols, plus *L* has been lowered a level so we need *1* extra symbol for each occurrence of *L*, resulting in a total of (*j+1)*P(l)+P(L)* extra symbols.

Obviously we should place the letter where it causes the least extra symbols in our output. Thus we have to calculate these values for all possible places, and choose the best option.

This is still not the best algorithm we could construct this way, because it does not take into account that dividing an existing node creates many empty nodes, which are potentially better places for future letters. Constructing an appropriate "weight" however for this effect is a difficult task, so we will not get into that. Instead, we will try to model this in another way.

Let us assign *n-1* letters at a time. In each step, we will gather *n-1* letters from the unassigned ones, choose where to put them, by calculating the weight of each possible place in a similar weight, and assign them there. This means that we do not need to weight empty nodes, because we will only add *n-1* to a "full" tree, which will remain a "full" tree. At the end, there may be extra letters left, if the total number of letters is not *n+i(n-1)* for some *i*. In this case, we will assign the rest of the letters one by one, as before.

This is basically the same algorithm as if we would have weighted all empty nodes at 0. If we would have done so, then after breaking a leaf node into a sub-tree, the next *n-2* letters (a total of *n-1*) would have been assigned to the empty nodes created. So we would assign letters not one by one, but *n-1* by *n-1*.

We implemented both versions of this method, and tested them extensively. Table 3 shows the test results we got on the same input text as we used before.

Table 3. Pushdown method

TEXT LENGTH	BINARY	3-ARY	4-ARY	5-ARY
ORIGINAL	364998	243332	182499	182499
HUFFMAN	260087	166335	132717	115893
PUSHDOWN	269426	177533	141245	124005
PUSHDOWN N-1	269426	168401	133356	115893

7. Conclusion

In this paper we began by introducing the problem of entropy based encoding in a general case, not only binary output. We introduced the Shannon-Fano coding for binary output. Next, we examined the possibility of generalizing the Shannon-Fano code for arbitrary output alphabet, in a similar manner, as one would generalize Huffman's code. We saw that there are some difficulties in doing so, for which we gave several solutions ("S-F MAX", "S-F MAX S", "S-F SUM", "S-F SUM S"). Afterwards we looked at possible improvements or alterations to the Shannon-Fano code. First we looked at non-ordering versions of the previously constructed algorithms, which yielded worse results than the ordered versions. Then we looked at alternatives to these algorithms, namely the distributing and pushdown methods ("S-F DISTRIBUTING", "S-F DISTRIBUTING S", "PUSHDOWN", PUSHDOWN N-1"). These algorithms performed well in our tests, but the best algorithm still seems to be "S-F SUM S".

Based on this judgement we ran a longer test, on a bigger input, with bigger alphabet, this time only with our "winner algorithm". Our randomly generated 100 paragraph text, which was used during testing for all the algorithms had 41 distinct letters, and had a size of 60 833 bytes, while the final test, ran only on the "S-F SUM S" method had 256 distinct letters, and had a size of 416 256 bytes. This test file was the executable file of the program itself which we used to test all our algorithm. Table 4 shows our test results.

Table 4. Final result

TEXT LENGTH	BINARY	3-ARY	4-ARY	5-ARY
ORIGINAL	3330048	2497536	1665024	1665024
HUFFMAN	2202062	1402926	1117106	963429
S-F SUM S	2207253		1136380	

Overall, we can conclude that this method is the most efficient generalization of the Shannon-Fano code for arbitrary output alphabets, and performs very closely to Huffman's code. The method however does have the disadvantage that it does not work on all inputs, and it runs slowly, because we used a brute force approach for finding the optimal cut. Subject of further research into the area could be the construction of a non-brute force algorithm for finding this optimum.

References

[1] Shannon CE: A mathematical theory of communication, Bell System Technical Journal, Vol. 27, pp 379-423, 1948

[2] Fano RM: The transmission of information, Technical report no. 65, Research Laboratory of Electronics, Massachusetts Institute of Technology, 1949

[3] Huffman D: A method for the construction of minimum-redundancy codes, Proceedings of the IRE, Vol. 40, No. 9, pp 1098-1101, 1952

[4] Cover TM, Thomas JA: Elements of information theory, 2nd edition, New Jersey, John Wiley & Sons, Inc., 2006, ISBN: 9780471241959

[5] Jones GA, Jones JM: Information and coding theory, London, Springer, 2000, ISBN: 9781852336226

[6] Yeung RW: Information theory and network coding, London, Springer, 2008, ISBN: 9780387792330

[7] Sayood K: Introduction to data compression, 2nd edition, London, Academic Press 2000, ISBN: 9781558605589

[8] Sayood K: Lossless compression handbook, London, Academic Press 2003, ISBN: 9780126208610

[9] Witten IH, Moffat A, Bell TC: Managing gigabytes: compressing and indexing documents, and images, 2nd edition, London, Academic Press, 1999, ISBN: 9781558605701

[10] Salomon D: Data compression: The complete reference, London, Springer, 1997, ISBN: 9780387982809

[11] Salomon D, Motta G: Handbook of data compression 5th edition, London, Springer, 2010, ISBN: 9781848829022

[12] Drozdek G: Elements of data compression, Pacific Grove, Brooks/Cole publishing, ISBN: 9780534384487