Ibn Sina School for Computer Science

Continiuity Project Report

Data Compression: Theory and Demonstration

Authors:
Ahmed Jazzar
Bassam Jaber
Othman Alkhamra

Supervisor:
Prof. Munther Dahleh

December 20, 2013

Abstract

In computer science and information theory, data compression involves encoding information using fewer bits than the original representation. The process of reducing the size of a data file is popularly referred to as data compression, although its formal name is source coding (coding done at the source of the data before it is stored or transmitted).

The main concepts of information theory can be grasped by considering the most widespread means of human communication: language. Two important aspects of a concise language are as follows: First, the most common words (e.g., "a", "the", "I") should be shorter than less common words (e.g., "roundabout", "generation", "mediocre",) so that sentences will not be too long. Such a trade-off in word length is analogous to data compression and is the essential aspect of source coding. Second, if part of a sentence is unheard or misheard due to noise - e.g., a passing car - the listener should still be able to glean the meaning of the underlying message. Such robustness is as essential for an electronic communication system as it is for a language; properly building such robustness into communications is done by channel coding. Source coding and channel coding are the fundamental concerns of information theory.

Coding theory is concerned with finding explicit methods, called codes, of increasing the efficiency and reducing the net error rate of data communication over a noisy channel to near the limit that Shannon proved is the maximum possible for that channel. These codes can be roughly subdivided into data compression (source coding) and error-correction (channel coding) techniques. In the latter case, it took many years to find the methods Shannon's work proved were possible. A third class of information theory codes are cryptographic algorithms (both codes and ciphers). Concepts, methods and results from coding theory and information theory are widely used in cryptography and cryptanalysis. See the article ban (information) for a historical application.

Compression is useful because it helps reduce resources usage, such as data storage space or transmission capacity. Because compressed data must be decompressed to use, this extra processing imposes computational or other costs through decompression; this situation is far from being a free lunch. Data compression is subject to a space-time complexity trade-off. For instance, a compression scheme for video may require expensive hardware for the video to be decompressed fast enough to be viewed as it is being decompressed, and the option to decompress the video in full before watching it may be inconvenient or require additional storage. The design of data compression schemes involves trade-offs among various factors, including the degree of compression, the amount of distortion introduced (e.g., when using lossy data compression), and the computational resources required to compress and uncompress the data.

In this project, we explore the foundations of data compression. The project was divided into four milestones, each milestone accumulated its proceeded milestone.

1 Milestone I

In this milestone, we generated samples of discrete random variable with a given probability mass function (pmf). We use Matlab as a simulation environment to help us in computing random samples of a continuous random variable that is uniformly distributed over [0,1].

1.1 Description

In information theory the discrete source is represented as a random variable with finite number of elements in the sample space (alphabets), and sampling the discrete random variable is used to generate the data independently . In general coding is concerned with finding a unique 1-1 mapping between the alphabets and sequences of 1âĂŹs and 0âĂŹs , so that a source with N alphabets can be encoded with LOG(N) bits ,so in the context of data compression we should avoid encoding all alphabets with the same length code instead we should assign the shortest code with the one that have the highest probability so that we can represent the source code with a less shorter sequence that needed for fixed encoding . So in the first milestone we have generated samples of discrete random variable with a given probability mass function (pmf) using Matlab environment , and We have considered three pmfs defined on an alphabet of size 8:

$$p_1(x) = \begin{cases} 1/8 & x=1 \\ 1/8 & x=2 \\ 1/8 & x=3 \\ 1/8 & x=4 \\ 1/8 & x=5 \\ 1/8 & x=6 \\ 1/8 & x=7 \\ 1/8 & x=8 \end{cases} \quad p_2(x) = \begin{cases} 1/16 & x=1 \\ 1/16 & x=2 \\ 5/16 & x=3 \\ 4/16 & x=4 \\ 2/16 & x=5 \\ 2/16 & x=6 \\ 0 & x=7 \\ 1/16 & x=8 \end{cases} \quad p_3(x) = \begin{cases} 0 & x=1 \\ 0 & x=2 \\ 1/4 & x=3 \\ 1/4 & x=4 \\ 0 & x=5 \\ 1/4 & x=6 \\ 1/4 & x=7 \\ 0 & x=8 \end{cases}$$

1.2 Objective

The objective was to write a Matlab function that provides independent samples of a general discrete random variable.

1.3 Implementation

The following Matlab code is responsible of generating the independent samples:

```
function S= DiscrInvTrans(x,p,ns)
    Y = cumsum(p);
    for i = 1:1:ns
        u = rand;
        k = 1;
          while u > Y(k)
```

```
            k = k+1;
        end
        S(i) = x(k);
        end
end
```

Matlab Code I: Independent Samples generator.

And we wrote three cells that describes each pmf:

```
%%
 ns = input('Enter the number of Samples : \n ');
X = 1:1:8;
p = [1/8 1/8 1/8 1/8 1/8 1/8 1/8 1/8];
Y= DiscrInvTrans(X,p,ns);
hist(Y,X);
xlim([0.5 8.5]);
xlabel('Alphabet');
ylabel('Percentage of occurrence');
```

Matlab Code II: pmf1 Implementation

```
%%
 ns = input('Enter the number of Samples : \n ');
X = 1:1:8;
p = [1/16 1/16 5/16 4/16 2/16 2/16 0 1/16];
Y= DiscrInvTrans(X,p,ns);
hist(Y,X);
xlim([0.5 8.5]);
xlabel('Alphabet');
ylabel('Percentage of occurrence');
```

Matlab Code III: pmf2 Implementation

```
%%
 ns = input('Enter the number of Samples : \n ');
X = 1:1:8;
p = [0 0 1/4 1/4 0 1/4 1/4 0 ];
Y= DiscrInvTrans(X,p,ns);
hist(Y,X);
xlim([0.5 8.5]);
xlabel('Alphabet');
ylabel('Percentage of occurrence');
```

Matlab Code IV: pmf3 Implementation

1.4 Sample Output

we include a Sample output for independent samples from the Matlab code described above for each pmf. We put the alphabet on x-axis, and the percentage of occurrences on y-axis.

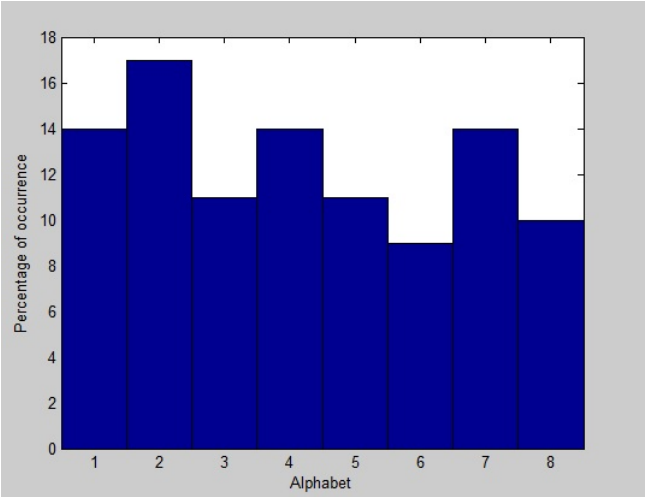


Image I: pmf1 Sample output.

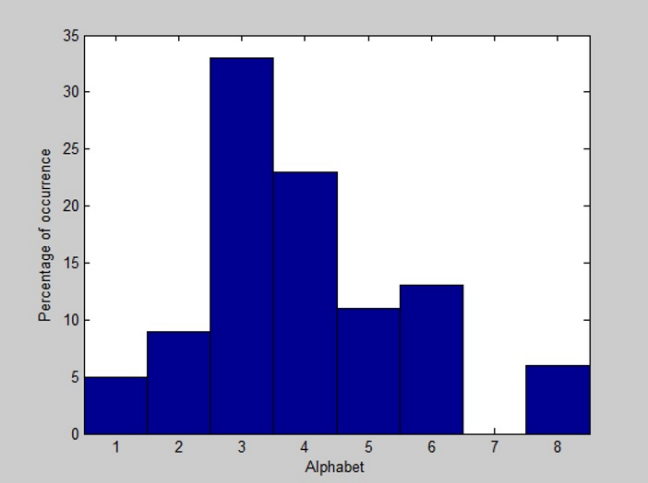


Image II: pmf2 Sample output.

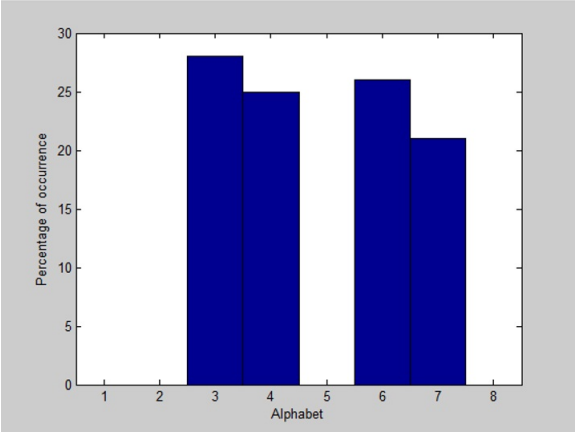


Image III: pmf3 Sample output.

1.5 Conclusions

The result of performing the same experiment a large number of times; the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed. And this is what the law of large numbers in probability theory describes.
The following histogram shows that generating 100,000 samples of pmf2 make the occurrence of each symbol reachs the probability of that symbol.

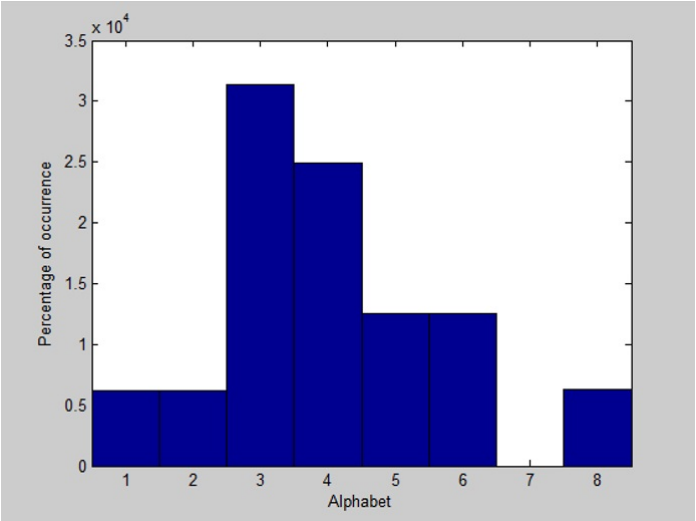


Image IV: pmfs 100,000 samples

2 Milestone II

2.1 Code Related Deliverables

In this part of the project, we are dealing with prefix codes or instantaneous codes, which state that no codeword is a prefix of any other codeword. This coding technique is self punctuated so we don't have to refer to the next codeword because it has an immediate recognizable end. For Example, a code {8, 85, 852} is not a prefix code, because codeword "8" is a prefix of both "85" and "852". In another hand, a code {6, 54, 369} is a prefix code because it's consensus the definition.

2.1.1 Code analysis

For any prefix code, it is important to map each symbol uniquely to a codeword, and to be obvious in decoding.
Below, we list the code that we want to analyze according to the previous properties. After it we put the major points we have on it.

$$1 \rightarrow 1101101$$
$$2 \rightarrow 1101100$$
$$3 \rightarrow 0$$
$$4 \rightarrow 10$$
$$5 \rightarrow 111$$
$$6 \rightarrow 1100$$
$$7 \rightarrow 1101101$$
$$8 \rightarrow 11010$$

- $\forall$ symbols i, j $\in$ alphabets {1, 2, 3, 4, 5, 6, 7, 8},So that $X_i \neq X_j$ and $i \neq j$, also $C(x_i) \neq C(x_j)$, we can say that both codes are non-singular since every element in range x is mapped into different String in D* . and it satisfies the non-singular definition, which states:

  Theorem 1. *A code is said to be nonsingular if every element of the range of X is mapped into a different string in D*; that is,*
  $X \neq X' \rightarrow C(X) \neq C(X')$

  So we can say that both code 1 and code 2 apply this property .

- For fixed length encoding, it is straight forward to divide the sequence of code words and encode them to related symbols depending on the length of a single code word. For code 2 in order to prove this property we need to prove that the code is uniquely decodable.

  Theorem 2. *The extension C* of a code C is the mapping from finite length strings of X to finite-length strings of D, defined by :*
  $C(x_1x_2...x_n) = C(x_1)C(x_2)...C(x_n)$

Theorem 3. *A code is called uniquely decodable if its extension is nonsingular.*

Referring to the previous definitions , We can say that code 2 is uniquely decodable because the mapping from the alphabets to the codes which called extension of the code is non-singular.

2.1.2 GENERATING PREFIX CODE

Theorem 4. *A code is called a prefix code or an instantaneous code if no code word is a prefix of any other code word.*
$\forall$ alphabet let X_i Denotes the alphabet 1, . . . 8 and $C(X_i)$ denotes the corresponding code word, we can conclude that for all alphabets $C(X_i)$ is not a prefix of any other $C(X_j)$ where $j \neq i$
Th following table includes all prefixes of code 2.

Codeword	prefixes
1101101	1, 11 ,110, 1101, 11011, 110110, 1101101
1101100	1, 11 ,110, 1101, 11011, 110110, 1101100
0	0
10	1 , 10
111	1 , 11 , 111
1100	1 , 11, 110 , 1100
110111	1 , 11 , 110 , 1101 , 11011, 110111
11010	1 , 11 , 110, 1101 , 11010

Table 1. prefixes of the codewords

As we see, no code word of the code 2 is a prefix of any other code word in the alphabets , so the code is a prefix code.

2.2 COMPUTATIONS

2.2.1 SAMPLE MEAN

Sample mean in general is the Average Length of the Code, so we simply adding the length of all codewords and dividing by the number of samples.

- For the first code, we can notice that the average length is 3, since all the codewords have the same length, which is 3.
- For the Second one, the Average length is (7+7+1+2+3+4+6+5) / (8) = 4.375 Bits.

The results above shows that the first code produces a smaller value than the second one. In order to do more computations,we derived another instantaneous code, which is:

$$1 \to 0$$
$$2 \to 10$$
$$3 \to 110$$
$$4 \to 1110$$
$$5 \to 11110$$
$$6 \to 111110$$
$$7 \to 1111110$$
$$8 \to 11111110$$

after doing the same computations on it, we got that its average length equals 4.5, which is as the same as the average length of the second code.

Theorem 5. *The expected length $L(C)$ of a source code $C(x)$ for a random variable X with probability mass function $p(x)$ is given by*
$L(C) = \sum_{x' \in X} P(x)l(x)$

The table below shows the expected length for each code in different pmfs:

	pmf1	pmf2	pmf3
Code 1	3	2.87	3
Code 2	4.5	1.8	3.5
Code 3	4.5	1.8	5

Comparing the expected lengths in average with the sample averages we calculated earlier:

- The first code is approximately around the sample mean.
- The second one is less than the sample mean.
- The third one is less the sample mean too.

2.3 Conclusions

- The average length of the code does not depend on the probability of occurrence of each symbol.
- And also the average code length does not depend on the pmf.
- The expected length of each code depends on the pmf if the pmf is not fixed.
- Variable code length becomes a nice coding method if we know how to reduce the expected length of the code.
- We think that if we reduce length of the most occur codewords, the expected length becomes smaller.

3 Milestone III

3.1 Kraft's Inequality

3.1.1 Description

Kraft's inequality gives a sufficient condition for the existence of a prefix code for a given set of codewords length. It states that the length of the code must satisfy the following condition for an alphabet of size D, and a codewords of lengths l_i.

$$\sum_i D^{-l_i} \leq 1$$

PROOF

We will prove that any codeword set that satisfies the prefix condition has to satisfy the Kraft inequality and that the Kraft inequality is a sufficient condition for the existence of a codeword set with the specified set of codeword lengths. To do so, we have a D-ary tree of the code, which we can get from it the codewords of that code. And as we mentioned before, for a prefix code there is no valid codeword in the system that is a prefix of any other valid codeword in the set. In the following tree, every codeword must be a leaf in the code tree - no codeword is an ancestor of another codeword - to have a prefix code.

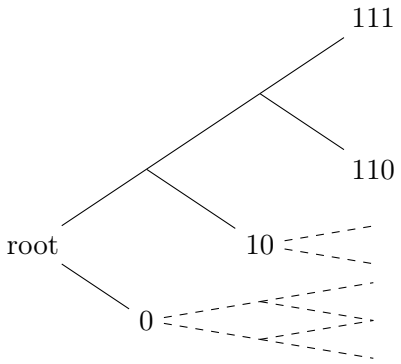


FIGURE 1 Code Tree for Craft Inequality

Suppose that $\ell_1 \leq \ell_2 \leq ... \leq \ell_{max}$. Let A be the full D-ary tree of depth ℓ_{max} . Every word of length $\ell \leq \ell_{max}$ over a D-ary alphabet corresponds to a node in this tree at depth ℓ. The ith word in the prefix code corresponds to a node v_i; let A_i be the set of all leaf nodes in the subtree of A rooted at v_i . Clearly

$$|A_i| = D^{\ell_{max} - \ell_i}.$$

Since the code is a prefix code,

$$A_i \cap A_j = \Phi, \quad i \neq j.$$

Thus, given that the total number of nodes at depth ℓ_{max} is $D^{\ell_{max}}$,

$$|\bigcup_i A_i| = \sum_i D^{\ell_{max}-\ell_i} \leq D^{\ell_{max}}$$

from which the result follows.

Conversely, given any ordered sequence of max natural numbers,

$$\ell_1 \leq \ell_2 \leq \cdots \leq \ell_{max}$$

satisfying the Kraft inequality, one can construct a prefix code with codeword lengths equal to ℓ_i by pruning subtrees from a full D-ary tree of depth ℓ_{max} . First choose any node from the full tree at depth ℓ_1 and remove all of its descendants. This removes $D^{-\ell_1}$ fraction of the nodes from the full tree from being considered for the rest of the remaining codewords. The next iteration removes $D^{-\ell_2}$ fraction of the full tree for total of $D^{-\ell_1} + D^{-\ell_2}$. After it iterations,

$$\sum_i^{it} D^{-\ell_i}$$

fraction of the full tree nodes are removed from consideration for any remaining codewords. But, by the assumption, this sum is less than 1 for all $it < max$. Thus prefix code with lengths ℓ_i can be constructed for all max source symbols.

3.2 Optimal Code Length problem

The main idea of this deliverable is to explain the optimal code length problem. Hence, we can say that a code to be optimal (for a given probability distribution) if no other code with a lower mean codeword exists.
It's clear that Optimal Code Length problem is the problem of finding the prefix code with minimum expected length.
Mathematically:
we want to minimize

$$L = \sum p_i l_i$$

over all codeword lengths, satisfying

$$\sum D^{-l_i} \leq 1.$$

A way to do this, is using Lagrange multiplier as the minimization of

$$\sum p_i l_i + \lambda(\sum D^{-l_i}).$$

After differentiating this equation and solve it, we will have this formula

$$-\sum p_i \ log_D \ p_i$$

which equals to

$$H_D(X)$$

To prove it, suppose that we code one symbol at a time, an optimal code satisfies

$$l^\star < H_D(X) + 1$$

let

$$l_i = \lceil -\log_D P_i \rceil$$

We have that

$$\begin{gathered} -\log_D P_i \leq \lceil -\log_D P_i \rceil < -\log_D P_i + 1. \\ \sum D^{-l_i} = \sum D^{-\lceil -log_D p_i \rceil} \\ \leq \sum D^{log_D p_i} \\ = \sum p_i = 1 \end{gathered}$$

We can notice now that Kraft's inequality satisfied, therefore a code with the given codeword lengths exists.
Hence, the expected length

$$L \geq H_D(X).$$

3.3 Entropy

We notice from the last section that the expected length depends on the entropy, which is a measure of uncertainty of a random variable. In our problem, we can define entropy as a smallest number of bits needed, on the average, to represent a symbol (the average on all the symbols code lengths). Hence, it is a lower bound on the average number of bits needed to represent the symbols (the data compression limit).
We can calculate the entropy mathematically using following low

$$H(X) = -\sum_{x \in \chi} p(x) \lg(p(x))$$

To explain how entropy random a source, suppose that we want to determine the value of X with the minimum number of binary questions. An efficient first question is "Is X = a?" This splits the probability in half. If the answer to the first question is no, the second question can be "Is X = b?" The third question can be "Is X = c?" The resulting expected number of binary questions required is 1.75. This turns out to be the minimum expected number of binary questions required to determine the value of X.
In the same we determined the value of the expected length L before.

3.3.1 Calculations

The following list contains the entropy of each pmf given in the project description.

	p1	p2	p3
H(X)	4.5	4	5

3.4 Conclusions

- In this part of the project, we notice that Kraft's inequality gives a restriction on the length of the codewords, so it'll looks like a pmf with a total measure less than or equals to one.
- the expected description length must be greater than or equal to entropy
- Entropy only deals with the pmf.

4 Milestone IV

4.1 Huffman codes construction

Huffman encoding: is an optimal encoding for constructing a prefix code with shortest expected length with a given distribution. For example, considering the following table:

Symbol	1	2	3	4	5
probability	0.25	0.25	0.2	0.15	0.15

Using Huffman encoding we expect that the longest codewords assigned to symbols with lowest probability so that we obtain the shortest average length , and also we need to obtain a prefix code that helps us to identify the symbols without any ambiguity.
So Huffman states that we combine the symbols with lowest probability and proceeding this way until we are finally left with one symbol with probability of 1 , then we start assigning codewords to symbols to obtain the Huffman code.
The following table illustrate the procedure of obtaining the Huffman code:

Codeword Length	Codeword	X	Probability
2	01	1	0.25 0.3 0.45 0.55 1
2	10	2	0.25 0.25 0.3 0.45
2	11	3	0.2 0.25 0.25
3	000	4	0.15 0.2
3	001	5	0.15

The number of symbols combined depend on the number of symbols in the alphabet used for encoding , so for binary systems which have only 0,1 we only combine 2 symbols with minimum probability .
In the same hand, considering the same random variable we want to obtain a ternary code 0,1,2.

Symbol	1	2	3	4	5
probability	0.25	0.25	0.2	0.15	0.15

For ternary code the procedure is the same in general , we combine the 3 symbols with minimum probability , but we may not have a sufficient number of symbols to combine them each time , let the number of symbols left to combine D , if we have D âĽě 3 , we may not have enough symbols to combine at each step , so we add a dummy symbol with probability of 0 and we use it only to complete the Huffman tree , so we want that the total number of symbol to be (1+k(D-1)) , where k is the number of merges.

Codeword	X	Probability			
1	1	0.25	0.25	0.5	1.0
2	2	0.25	0.25	0.25	
01	3	0.2	0.2	0.25	
02	4	0.1	0.2		
000	5	0.1	0.1		
001	6	0.1			
002	Dummy	0.0			

Huffman coding for weighted codewords. Huffman's algorithm for minimizing $\sum\ p_i * l_i$ can be applied to any set of numbers $pi \geq 0$ regardless of $\sum p_i$. In this case, the Huffman code minimizes the sum of weighted code lengths $\sum w_i * l_i$ rather than the average code length.
Huffman codes and Shannon codes , for some particular cases Shannon codes could be less than the codeword length of the corresponding symbol of an optimal Huffman code. Although either the Shannon code or the Huffman code can be shorter for individual symbols, the Huffman code is shorter on average ,Also the Shannon code and the Huffman code differ by less than 1 bit in expected code length (since both lie between H and H + 1).

4.2 Proof of Optimality

Considering an optimal code C_m:

- If $p_j > p_k$, then $l_j \leq l_k$. Here we swap codewords.
  Consider $C\prime_m$, with the codewords j and k of C_m interchanged. Then
  $L(C\prime_m)L(C_m) = \sum p_i l_i\prime - \sum p_i l_i$
  $= p_j l_k + p_k l_j$ - $p_j l_j$ - $p_k l_k$
  $= (p_j$ - $p_k)(l_k$ - $l_j)$.
  But p_j - $p_k > 0$, and since C_m is optimal, $L(C\prime_m)$ - $L(C_m) \leq 0$. Hence, we must have $l_j \leq l_k$. Thus, $C_m$itself satisfies property 1.

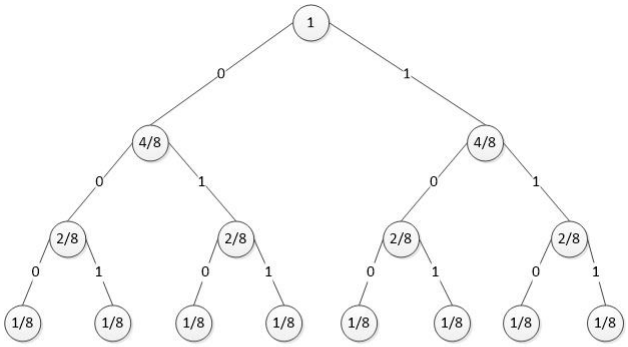
- The two longest codewords are of the same length. Here we trim the codewords. If the two longest codewords are not of the same length, one can delete the last bit of the longer one, preserving the prefix property and achieving lower expected codeword length. Hence, the two longest codewords must have the same length. By property 1, the longest codewords must belong to the least probable source symbols.

- The two longest codewords differ only in the last bit and correspond to the two least likely symbols. Not all optimal codes satisfy this property, but by rearranging, we can find an optimal code that does. If there is a maximal-length codeword without a sibling, we can delete the last bit of the codeword and still satisfy the prefix property. This reduces the average codeword length and contradicts the optimality of the code. Hence, every maximal-length codeword in

any optimal code has a sibling. Now we can exchange the longest codewords so that the two lowest-probability source symbols are associated with two siblings on the tree. This does not change the expected length, $\sum p_i l_i$. Thus, the codewords for the two lowest-probability source symbols have maximal length and agree in all but the last bit.

Summarizing, we have shown that if $p_1 \geq p_2 \geq ... \geq p_m$, there exists an optimal code with $l_1 \leq l_2 \leq ... \leq l_m 1 = l_m$, and codewords $C(x_m 1)$ and $C(x_m)$ that differ only in the last bit.

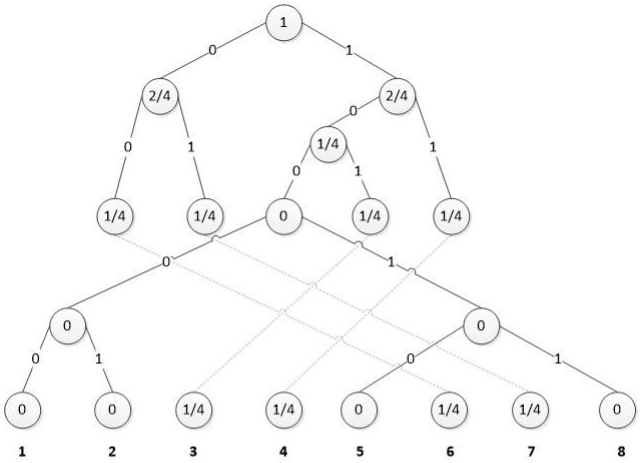
4.3 pmfs Huffman codes



X	C(X)
1	000
2	001
3	010
4	011
5	100
6	101
7	110
8	111

L(C) = 3*(1/8) + 3*(1/8) + 3*(1/8) + 3*(1/8) + 3*(1/8) + 3*(1/8) + 3*(1/8) + 3*(1/8) = 3

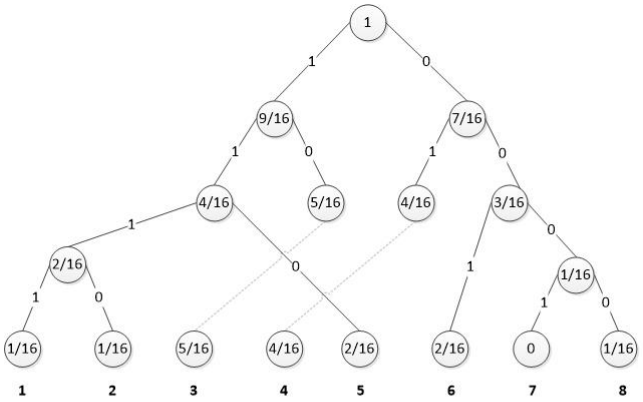
Image V: pmf1 Huffman Code



X	C(X)
1	10000
2	10001
3	101
4	11
5	10010
6	00
7	01
8	10011

L(C) = 5*(0) + 5*(0) + 3*(1/4) + 2*(1/4) + 5*(0) + 2*(1/4) + 2*(1/4) + 0*(5) = 2.25

Image VI: pmf1 Huffman Code



X	C(X)
1	1111
2	1110
3	10
4	01
5	110
6	001
7	0001
8	0000

L(C) = 4*(1/16) + 4*(1/16) + 2*(5/16) + 2*(4/16) + 3*(2/16) + 2*(2/16) + 4*(0) + 4*(1/16) = 2.5

Image VII: pmf1 Huffman Code

5 References

- Continuity Projects Description, Summer 2013, Ibn Sina School for Computer Science. 12 - 15.
- Thoms M. Cover, Joy A. Thomas (2006). Elements of Information Theory. A JOHN WILEY & SONS, INC., PUBLICATION. ISBN 10 0-471-24195-4. 103 - 127.
- Fazlollah M. Reza (1961, 1994). An Introduction to Information Theory. Dover Publications, Inc., New York. ISBN 0-486-68210-2.
- Wade, Graham (1994). Signal coding and processing (2 ed.). Cambridge University Press. p. 34. ISBN 978-0-521-42336-6. Retrieved 2011-12-22. "The broad objective of source coding is to exploit or remove 'inefficient' redundancy in the PCM source and thereby achieve a reduction in the overall source rate R."
- Mahdi, O.A.; Mohammed, M.A.; Mohamed, A.J. (November 2012). "Implementing a Novel Approach an Convert Audio Compression to Text Coding via Hybrid Technique". International Journal of Computer Science Issues 9 (6, No. 3): 53âĂŞ59. Retrieved 6 March 2013.
- Pujar, J.H.; Kadlaskar, L.M. (May 2010). "A New Lossless Method of Image Compression and Decompression Using Huffman Coding Techniques". Journal of Theoretical and Applied Information Technology 15 (1): 18âĂŞ23.
- Salomon, David (2008). A Concise Introduction to Data Compression. Berlin: Springer. ISBN 9781848000728.