ÇUKUROVA UNIVERSITY

INSTITUTE OF NATURAL AND APPLIED SCIENCES

PhD THESIS

Devising a Coding Mechanism for Compression Algorithms

Muhammed Mustafa AŞŞIK

Department of Computer Engineering

March, 2024

ÇUKUROVA UNIVERSITY

INSTITUTE OF NATURAL AND APPLIED SCIENCES

PhD THESIS APPROVAL

Devising a Coding Mechanism for Compression Algorithms

Muhammed Mustafa AŞŞIK

Department of Computer Engineering

This Doctorate Thesis was evaluated by the following Jury Members on …/…/2024 and was approved by unanimity / majority of votes.

Jury : Assoc. Prof. Dr. Fatih ABUT (Advisor) …….

: Prof. Dr. Ulus ÇEVİK …….

: Prof. Dr. Zekeriya TÜFEKÇİ ……

: Assoc. Prof. Dr. Mümine KAYA KELEŞ ……

: Asst. Prof. Dr. Esra SARAÇ ……

This Thesis was written in the Department of Computer Engıneering, Institute of Natural and Applied Sciences.

Thesis Number:

Prof. Dr. Sadık DİNÇER
Director
Institute of Natural and Applied Sciences

CONTENTS **PAGE**

CUKUROVA UNIVERSITY

INSTITUTE OF NATURAL AND APPLIED SCIENCES

PhD THESIS

Devising a Coding Mechanism for Compression Algorithms

Muhammed Mustafa AŞŞIK

Advisor: Assoc. Prof. Dr. Fatih ABUT

Department of Computer Engineering

ABSTRACT

Canonical Huffman Coding is a data compression algorithm still widely used in many applications. Originally, to produce canonical Huffman codes, first, the code tree is created, then from the leaves to the root, code lengths are obtained. Canonical codes are then calculated from these lengths. Instead of this multi-stage process, Algebraic Canonical Huffman Coding (ACHC), which calculates the code lengths in one stage through direct calculation, is proposed in this thesis. After the calculation of the lengths, the codewords corresponding to each length are calculated by binary addition. Time complexity is O(n) for ACHC and is O(n(logn+l)) for classical canonical Huffman coding. Space complexity is O(5n) standard for Huffman coding and O(n) for ACHC. This means memory saving is 80%. However, the average bit length per symbol is not usually optimal but is much closer to the optimal than similar methods. Therefore, the second algorithm (ES_ACHC) that optimizes the ACHC algorithm with the Evolutionary Strategies algorithm, is also proposed in this thesis. With this algorithm, optimum canonical codes were obtained with shorter loops. The adaptive application of the ACHC algorithm (A_ACHC) is the third contribution of this thesis to the existing literature. With the A_ACHC algorithm, 80% of memory savings have been achieved with the same compression ratio compared to the well-known Vitter algorithm.

Keywords: Canonical Huffman, Adaptive Huffman, Encoding, Evolution Strategies, Data Compression

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

DOKTORA TEZİ

Sıkıştırma Algoritması için Bir Kodlama Mekanizmasının Tasarlanması

Muhammed Mustafa AŞŞIK

Danışman: Assoc. Prof. Dr. Fatih ABUT

Bilgisayar Mühendisliği Anabilim Dalı

ÖZ

Kanonik Huffman Kodlama halen birçok uygulamada kullanılan bir olasılık tabanlı veri sıkıştırma algoritmasıdır. Geleneksel uygulamada Kanonik Huffman kodlarını üretmek için önce kod ağacı oluşturulur, daha sonra yapraklardan köke doğru gidilerek kod sözcükleri ve bunların bit sayısı (uzunlukları) elde edilir. Daha sonra bu uzunluklardan Kanonik kodlar hesaplanır. Bu çok aşamalı işlem yerine, bu tezde, doğrudan hesaplama yoluyla kod uzunluklarını tek aşamada hesaplayan Cebirsel Kanonik Huffman Kodlaması (ACHC) önerilmiştir. Uzunluklar hesaplandıktan sonra ikili toplama ile her uzunluğa karşılık gelen kod sözcükleri hesaplanır. Klasik yöntemin O(n(logn+l)) zaman karmaşıklığına karşılık, ACHC algoritması O(n) zaman karmaşıklığına sahiptir. Uzay karmaşıklığı standart Huffman kodlaması için O(5n) ve ACHC için O(n)'dir. Bu da %80 bellek tasarrufu anlamına gelmektedir. Ancak sembol başına ortalama bit uzunluğu genellikle en iyi değer olmayıp, en iyi değere benzerlerinden çok daha yakındır. Bu nedenle, bu tezde ACHC algoritmasını Evrimsel Stratejiler algoritması ile en iyileştiren ikinci algoritma (ES_ACHC) önerilmiştir. Bu algoritmayla kısa döngü zamanında optimum kanonik kodlar elde edilmiştir. ACHC algoritmasının adaptif uygulaması (A_ACHC), bu tezin mevcut bilgilere üçüncü katkısıdır. A_ACHC algoritması ile çok bilinen Vitter algoritmasına göre ortalama olarak aynı sıkıştırma oranı ile %80 bellek tasarrufu sağlanmıştır.

Anahtar Kelimeler: Kanonik Huffman, Adaptif Huffman, Kodlama, Evrimsel Stratejiler, Veri Sıkıştırma

GENİŞLETİLMİŞ ÖZET

Bilişim alanında insanoğlu sınırsız denebilecek devasa bir veri ile çalışmaktadır. Ancak bu verilerin depolanacağı bellek konumu ve bu verilerin iletileceği kanal kapasiteleri sınırlıdır. Sınırlı kaynaklarla sınırsız verinin nasıl kullanılabileceği sorusu Bilgi Teorisini ortaya çıkarmıştır. Bilgi Teorisi, bir veri yığınıyla oluşturulan bir mesajın taşıdığı bilgiyi kaybetmeden boyutunun ne kadar küçültülebileceğini açıklar (Shannon, 1948). Veri yığınının boyutunu, içerdiği bilgileri kaybetmeden azaltan işleme veri sıkıştırma denir. Veri sıkıştırma yöntemleri kullanılarak, bilgi kaybı olmadan veri bloklarının boyutu küçültülmüş, böylece bellek alanından ve iletişim süresinden tasarruf sağlanmıştır.

D. Huffman tarafından önerilen ve halen kullanılmakta olan Huffman Kodlaması, halen farklı araştırmacıların üzerinde çalıştığı önemli bir algoritmadır (Huffman, 1952). Görüntü sıkıştırma (JPEG), metin sıkıştırma (DEFLATE, GZIP gibi), ses sıkıştırma (MP3) Huffman kodlamanın önemli uygulama alanlarıdır (Hosseini, 2012). Ayrıca, sıkıştırılmış verinin kodunu çözmeden kod kelimelerine rasgele erişim yeteneği, sıkıştırılmış veri yapıları ve sıkıştırılmış metin veri tabanları gibi çok sayıda senaryo, bu kodlamayı önemli kılmaktadır (Navarro, 2013).

Huffman kodlamanın bir başka versiyonu da Kanonik Huffman kodlarıdır. Sembol başına aynı ortalama bit uzunluğuna sahip olmalarına rağmen, Kanonik Huffman kodlamasının standart Huffman kodlamasına göre bazı avantajları vardır.

Bu tezde, kanonik Huffman kodlarının uzunluklarını cebirsel olarak hesaplayan bir algoritma (Cebirsel Kanonik Huffman Kodlaması, ACHC) önerilmiştir. ACHC, klasik Kanonik Huffman kodlamasından daha basit ve daha kısa kod satırları gerektiren bir algoritmadır. Üretilen sembol başına ortalama bit uzunluğu genellikle en iyi değer değildir, ancak benzer algoritmalara göre en iyi değere daha yakındır. Sadece ağacı elde etmek için standart Huffman kodlamasının zaman karmaşıklığı O (nlogn)'dır. Uzay karmaşıklığı ise O(5n)'dir (Moffat, 2020). Ağacı elde ettikten sonra kanonik kod uzunluklarını belirlemek için gereken süre, tüm semboller için O (nl)'dir; burada l, her sembol için kökten yaprağa olan mesafedir. Dolayısıyla, toplam süre O(n(logn+l))'dir. Önerilen ACHC algoritmasında, ağırlık dizisinin sıralı olması koşuluyla, uzunlukları ve kullanılan bellek miktarını hesaplamak için gereken süre Θ (n)'dir. Zaman karmaşıklıklarının karşılaştırılması, AHCH algoritmasının geleneksel kanonik Huffman kodlamasından daha hızlı olduğunu ifade eder. Huffman kodlama için uzay karmaşıklığının O(5n) olduğu göz önüne alındığında, ACHC ile elde edilen bellek tasarrufu %80'dir.

Bu tezde önerilen ACHC algoritmasının amacı, en iyi veya en iyi değere yakın "prefix-free" kod uzunluklarını basit ve hızlı bir şekilde elde etmektir.

p_i olasılığına sahip bir s_i sembolüyle ilişkili kod sözcüğün uzunluğu aşağıdaki gibi hesaplanır:

$$l_i = \begin{cases} 1 & if\ \ p_i \geq 0.7 \\ R(-\log(p_i)) & otherwise \end{cases}$$

İşlem sırasında p_i küçüldükçe l_i'nin hesaplanmasında hata oluşur. Bu nedenle, ölçeklendirme gerekli hale gelir. Ölçeklendirme iki parametre kullanılarak yapılır: e_i ve k_i. k_i, işlem anına kadar hesaplanan kümülatif olasılık değerleridir: $k_i = \sum_{x=1}^{i-1} p_x$. e_i, işlem görmemiş olasılıkların toplamıdır: $e_i = 1 - k_i$. Böylece ölçeklenmiş haliyle aşağıdaki formül elde edilir:

$$l_i = \begin{cases} 1 & if\ \ p_i \geq 0.7 \\ R(\log(e_i\ /p_i)) & otherwise \end{cases}$$

e_i parametresi şu şekilde hesaplanır:

$$e_i = \begin{cases} e_i = 1 - \ k_{i-1} & if\ t_{m-1} \leq \frac{N_j}{2} \\ e_i = e_{i-1} & otherwise \end{cases}$$

Burada $N_j = 2^{l_i}$ ve $t_m = (2^{u_i - u_{i-1}} * \ t_{m-1}) - 1$ ve $u_i = d_j + l_i$. u_i, i'inci kod sözcüğünün uzunluğu. d_j'nin formülü şu şekildedir:

$$d_j = \begin{cases} d_j = d_{j-1} + 1 & if\ t_{m-1} \leq \frac{N_j}{2} \\ d_j = d_{j-1} & otherwise \end{cases}$$

Böylece, herhangi bir kod sözcüğünün uzunluğunu hesaplamak için tüm parametreler tanımlanmıştır. Uzunluklar hesaplandıktan sonra, kod sözcükler de kanonik biçimde kolayca belirlenir. Örneğin, U= (1,2,3,3) ise kod sözcükleri C= (0, 10,110,111) olacaktır.

İkinci olarak, ACHC algoritması tarafından elde edilen en iyi değere yakın kod uzunlukları, Evrim Stratejileri (ESs) adı verilen evrimsel bir algoritma kullanılarak optimize edilmiştir. Kullanılan ESs algoritması (1 + λ) – CSP-ES gösterimi ile açıklanabilir. Burada 1 sayısı, ata olan tek bir ebeveyni (μ = 1) tanımlar. Ata, λ kadar kopyalanarak çoğaltılır ve bu kopyalar mutasyona uğrar. Sonuç olarak, λ adet çocuk üretilir. Mevcut ata ve çocuklar arasından en iyi birey seçilir. Seçilen en iyi birey, bir sonraki neslin atası olur. CSP (Sabit Strateji Parametresi), strateji parametresinin (SP) evrimleşmediği anlamına gelir. SP sabittir, çünkü her nesildeki her birey için rastgele seçilir (1 veya -1). Bu nedenle kullanılan algoritmaya "sabit strateji parametreli evrim stratejileri" denilebilir. Uygunluk işlevi, sembol başına ortalama bit uzunluğudur. Seçim yöntemi elitizmdir. Döngü için sabit bir sayı alınırsa, algoritmanın genel zaman karmaşıklığı $O(n^2)$'dir. Yavru sayısı n uzunluğunda 5*n olduğundan, algoritmanın uzay karmaşıklığı da $O(n^2)$ bayttır. Burada n, mesajda kullanılan farklı sembollerin sayısıdır.

Bu tezin odak noktası, tez kapsamında geliştirilen ACHC algoritmasının uyarlanabilir versiyonu olan Adaptif Cebirsel Kanonik Huffman Kodlamasıdır (A_ACHC). A_ACHC algoritması daha az program satırı gerektirir, bu nedenle programlaması daha kolaydır ve daha az diziye ve değişkene ihtiyaç duyduğu için daha az bellek gerektirir, bu da enerji tasarrufuna katkıda bulunur. Bu nedenle, A_CKHK algoritmasının metin ve medya sıkıştırma için olduğu kadar kablosuz sensör

ağları ve giyilebilir cihazlar gibi enerji tasarruflu mobil cihazlar için kullanılması, diğer adaptif Huffman algoritmalarına göre daha uygun bir çözüm olabilir.

Adaptif Cebirsel Kanonik Huffman Kodlama (A_ACHC) algoritması ACHC algoritmasına adaptif sıkıştırma yöntemi uygulanarak, elde edilir. Adaptif sıkıştırma yöntemi şu anlama gelir: "Kodlanacak karakter mevcut kod tablosunda varsa, tabloya göre kodlayın, aksi takdirde ASCII kodunu iletin ve tabloyu güncelleyin". A_ACHC algoritması da aynı prensipte çalışır. Başlangıçta sadece henüz iletilmemiş karakteri temsil eden * karakterini içeren tablo ile başlar. Mesajdan bir karakter alındığında, eğer kod tablosunda yer almıyorsa önce * karakterinin uzunluğuna karşılık gelen kod, ardından ASCII kodu iletilir. Karakter ve ağırlığı tabloya eklenir, ardından tablodaki kod uzunlukları ACHC algoritmasına göre güncellenir. Karakter tabloda ise tablodan karşılık gelen uzunluğa göre belirlenen kod iletilerek karakterin ağırlığı bir artırılır ve tablodaki kod uzunlukları yeniden hesaplanarak güncellenir. Alfabe boyutu n iken m önceden işlenmiş karakterlerin sayısı ise, bir karakteri kodlamak için gereken zaman karmaşıklığı O(n-m)'dir. A_ACHC için gereken bellek miktarı diğerlerinden daha azdır. Uzay karmaşıklığı, n adet sembol ve t ağırlığı için $2n\log_2(n) + 8n + n\log_2(t)$ bittir. Vitter algoritması ile karşılaştırıldığında, n arttıkça bellek tasarrufu ortalama %80'e yaklaşır. Bu, A_ACHC algoritması için önemli bir avantajdır.

ACHC algoritması çok basit ve uygulanması kolaydır. Hızı, basitliği ve kullandığı bellek miktarı göz önüne alındığında çeşitli uygulamalarda tercih edilebilir. A_ACHC algoritması kullanılarak aynı sıkıştırma oranları sağlanırken diğer algoritmalara göre %80 daha az bellek kullanımı sağlanabileceği gösterilmiştir. Hafıza tasarrufu ve daha kısa kod uygulaması nedeniyle A_ACHC algoritmasının kablosuz sensör ağlarında kullanılan sensörler ve giyilebilir cihazlar için daha uygun olduğu düşünülmektedir.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my first advisor, Assoc. Prof. Dr. Mustafa ORAL, for his great effort, help, support, valuable advice and guidance throughout this thesis. I would like to thank my valuable advisor, Assoc. Prof. Dr. Fatih ABUT, who advised me in the last period my thesis due to the retirement of my first advisor and contributed its completion and Asst. Prof. Dr. Mehmet Sarıgül for his valuable help.

I would like to thank Prof. Dr. Ulus ÇEVİK, Prof. Dr. Zekeriya TÜFEKÇİ, Assoc. Prof. Dr. Mümine KAYA KELEŞ and Asst. Prof. Dr. Esra SARAÇ for accepting to take part in the examining committee of my thesis.

Finally, I would also like to thank my family for their love and encouragement throughout the whole process.

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

SYMBOLS AND ABBREVIATIONS

n : The number of the symbols in an alphabet

A : An event A

$P(A)$: The probability of event A

$i(A)$: Self information of event A

S : Set of symbols creating a message, alphabet, $S = \{s_1, s_2, \dots, s_n\}$

F : Set of frequencies of symbols, F=$\{f_1, f_2, \dots, f_n\}$

C : Codeword set, $C = \{c_1, c_2, \dots, c_n\}$

P : Probability Set of symbols, P=$\{p_1, p_2, \dots, p_n\}$

W : Weight set of Symbols, W=$\{w_1, w_2, \dots, w_n\}$

s_i : i^{th} element of the S set

M : Message carrying information

$H(M)$: Average self-information of M Message, Entropy of M Message

A_v : Average bit length of per symbol

R : Redundancy

KM : Kraft-MacMillan inequality

$O(\ )$: Time or space complexity, Big O, upper bound on the growth rate of the function

$\Theta(\ )$: Time or space complexity, Big Theta, bounded from the upper and lower

X_n : n-dimensional individual

SP_n : n-dimensional Strategy Parameter

C_n : n-Dimensional Chromosome for Evolutional Strategies, $C_n = (X_n, SP_n)$

Z : Normally Distributed Random Vector

g : Number of Generation in Evolution Strategies

μ : Number of Parents in Evolution Strategies

λ : Number of Children in Evolution Strategies

d_j : The length or depth from the root node to a reference node in a Huffman tree

l_i : The length or depth from a reference node to a leaf in a Huffman tree

u_i : The length or depth from the root node to the leaf in a Huffman tree, $u_i = d_j + l_i$

N_j : The number of leaves and nodes at distance d_j from any reference node

t_m : The position of the symbol s_i at a distance d_j

k_i : The cumulative probability values calculated up to the moment of operation

e_i : The sum of the remaining probabilities after s_{i-1} , $e_i = 1 - k_i$

ACHC : Algebraic Canonical Huffman Coding

AHC : Adaptive Huffman Coding

A_ACHC : Adaptive Algebraic Canonical Huffman Coding

ESs : Evolution Strategies

EA : Evolutional Algorithm

SP : The Strategy parameter used in Evolution Strategies

CR : Compression Ratio

CF : Compression Factor

KM : Kraft-MacMillan inequality

DWT : Discrete wavelet transform

DCT : Discrete cosine transform

JPEG : An image file format

MPEG : A video file format

MP3 : An audio file format

RLE : Run-length encoding

GA : Genetic algorithm

GP : Genetic programming

SA : Simulated annealing

FGK : Faller-Gallager-Knuth algorithm

NYT : Not yet transmitted character

ASCII : American Standard Code for Information Interchange

SH, SHA : Standard Huffman Encoding

CH : Canonical Huffman Encoding

CSP : Constant Strategy Parameter

ES_ACHC: Algebraic Canonical Huffman Coding obtained by Evolution Strategies

MP : Mutation Points

MR : Mutation Rate

1. INTRODUCTION

Resources such as water, soil, and mines given to human beings are limited. However, human needs are unlimited. The need to meet the unlimited needs of human beings with limited resources has given birth to the science of economics. Similarly, in the informatics area, human beings operate with huge data that can be said to be unlimited. However, the memory location where the data will be stored and the channel capacities to which the data will be transmitted are limited. The question of how unlimited data can be used with limited resources has revealed information theory. Information theory describes how much a message created with a data block can be reduced in size without losing the information it carries (Shannon, 1948). The process that reduces the size of the data block without losing the information it contains is called data compression. By using data compression methods, the size of data blocks is reduced without losing information, thus saving memory space and communication time. Table 1.1. a. shows a sample of the uncompressed raw data. This data occupies a total of 1.000.000 bits of memory, 8 bits per symbol. Table 1.1.b. represents the compressed version of the same data. Compressed data occupies 201.250 bits of memory, with an average of 1,61 bits per symbol. This saves 798.750 bits of space.

Many researchers have developed numerous algorithms for data compression in fields such as text, images, audio, and video. One of the developed algorithms is Huffman coding.

Huffman Coding, proposed by D. Huffman and still in use, is an important algorithm that different researchers are still working on (Huffman, 1952). Researchers have been conducting various studies to improve the performance of Huffman Coding and to produce codes differently from the traditional method (Moffat and Katajainen, 1995) (Moffat and Turpin, 1998) (Geldreich, 1999) (Polar, 2010) (Dube and Beaudoin, 2008) (Gajjala et al, 2020).

Table 1.1. Example of compressed and uncompressed data

a. uncompressed raw data

Symbols	Length (bits)	Number	Bits
00001110	8	125	1.000
10011100	8	1.125	9.000
01110111	8	11.250	90.000
00101011	8	50.000	400.000
10010110	8	62.500	500.000
Total		125.000	1.000.000
Bits Per Symbol:	8 bits		

(a)

b. compressed data

Symbols	Length (bits)	Number	Bits
1111	4	125	500
1110	4	1.125	4.500
110	3	11.250	33.750
10	2	50.000	100.000
0	1	62.500	62.500
Total		125.000	201.250
Bits Per Symbol:	1,61 bits	Saving:	79,88%

(b)

Because of their simplicity and speed, Huffman encoders are still used in many compression fields. Image compression (JPEG), text compression (like DEFLATE, GZIP), and audio compression (MP3) are important application areas of Huffman coding (Hosseini, 2012). In addition, numerous scenarios, such as the ability to randomly access code words without decoding compressed data, compressed data structures, and compressed text databases, make this encoding important (Navarro, 2013).

Another version of Huffman coding is the Canonical Huffman code. Although they have the same average bit length per symbol, Canonical Huffman encoding has some advantages over standard Huffman encoding.

Arithmetic coding is another probability-based encoder similar to the Huffman encoder. Although the compression ratio of the arithmetic encoder is better than that of the Huffman encoder, the Huffman encoder is much faster than the arithmetic encoder. In addition, Huffman coding is much easier to implement than Arithmetic Coding (Shahbahrami et al., 2011). For these reasons, the Huffman encoder can be preferred over the arithmetic encoder.

Because of reasons such as the widespread use of Huffman codes, their speed, and ease, researchers have conducted various studies to produce Huffman codes or Huffman-like codes using different algorithms. Here, the expression Huffman codes means codes with the optimum "prefix free" feature, while the expression Huffman-like means codes with "prefix free" feature close to the optimum. The expression "prefix-free" indicates that no codeword is a prefix of another codeword in the set of codewords used (Hromada ve Grošek 2022).

In this thesis, a new algorithm (Algebraic Canonical Huffman Coding, ACHC) that calculates algebraically the lengths of canonical Huffman codes is proposed. ACHC is a simpler algorithm that requires shorter code lines than classical Canonical Huffman coding. The average bit length per generated symbol is often not optimal but is closer to the optimal value than similar algorithms. The time complexity of standard Huffman coding is O ($nlogn$) just to obtain the tree. Space complexity is O ($5n$) (Moffat, 2020). The time required to determine the canonical code lengths after obtaining the tree is O (nl) for all symbols, where l is the distance from the root to the leaf for each symbol. Hence, the total time is O (n ($logn + l$)). In the proposed ACHC algorithm, the time required to calculate the lengths and the amount of memory used is Θ (n), provided that the weight array is sequential. Comparing the time complexities indicates that the AHCH algorithm is faster than traditional canonical Huffman coding. Furthermore, given that the space complexity for Huffman coding is O(5n), the memory savings achieved with ACHC is 80%.

The near-optimal code lengths obtained by the ACHC algorithm have been optimized using an evolutionary algorithm called Evolution Strategies (ESs). ESs are based on selecting the best offspring from mutated offspring from an ancestor using a fitness function. Children are mutated by an evolving function called the strategy parameter (SP). However; SP, which should evolve as a feature of ESs, does not evolve in the model used and is randomly selected (either1 or -1) for each

individual in each generation. For this reason, the algorithm used can be called the "Evolution Strategies with Constant Parameter". In addition, the mentioned algorithm in question is an application of the sexless reproduction model.

The ACHC algorithm is a semi-static algorithm and is used to compress sources for which probability values are previously known. Semi-static encoders first scan a source, for example, a text file, to detect the probabilities of the symbols, and then compress the file according to the detected probabilities. Traditional Huffman coding is an example of this (Moffat ve Petri, 2020). However, semi-static encoders cannot be used in instantaneous situations where the data are constantly changing. Because in this case, the probability distribution is unpredictable. In addition, two passes over the data increase the compression time. As a solution to these negative situations, adaptive (dynamic) compression algorithms have emerged. In the adaptive application, the character to be compressed is compressed according to the probability distribution of the preceding characters. After the character is compressed, the probability distribution table is updated. Adaptive algorithms are one-pass algorithms because the data compression process is performed at one time. Adaptive Huffman coding (AHC) is an example of this type (Sayood, 2018). AHC is currently used in various applications, such as text compression, media compression, mobile sensors, and data encryption.

While applying data encryption processes, compression is applied as well as encryption. In some encryption algorithms, encryption is also applied when compressing text with AHC (Usama et al., 2021). In wireless sensor networks, the information coming from the sensors and the information from the wearable devices (Klus et al, 2020) are also compressed, thus saving energy and providing transmission in a shorter time. AHC is mostly used for compression (Kalaivani and Tharini 2020).

The main focus of this thesis is adaptive algebraic canonical Huffman coding (A_ ACHC), the adaptive version of the ACHC algorithm developed within the thesis. The A_ACHC algorithm requires fewer program lines, so it is easier to program and requires less memory, as it requires fewer arrays and variables, resulting in 80% energy savings. Therefore, using the A_CKHK algorithm for text and media compression, as well as for energy-saving mobile devices such as wireless sensor networks and wearables, may be a more suitable solution than other adaptive Huffman algorithms. Both ACHC and A_ACHC have shorter lines of code than SHC and Vitter's algorithm and are more understandable and easier to program. Code examples are given in the Appendix.

The organization of this thesis is as follows:

In the first chapter, the subject of the thesis is introduced, and the advantages of the algorithms discussed in the thesis are explained by comparing them with similar algorithms.

The previous works on the three algorithms proposed in the three sections constitute the subject of the second chapter.

Detailed information about the previously developed compression algorithms related to the thesis topic, knowledge about the "Information Theory ", which is the basis of the data compression

branch, and the important definitions used in data compression are given in the third chapter. In addition, the Calgary Corpus used in the tests is introduced in this chapter.

In the fourth chapter, the three algorithms proposed and discussed within the thesis are explained in three sections.

The fifth chapter compares the proposed algorithms with the traditional algorithms and discusses the test results. The sixth chapter covers future work and a general evaluation of the algorithms discussed.

2. PRELIMINARY WORK

2.1. Introduction

In this thesis, three interrelated data compression algorithms are proposed. Therefore, previous studies on these algorithms are described under three headings.

2.2. Previous Works on Standard and Canonical Huffman Encoding

Due to reasons such as the prevalence, speed, and ease of Huffman codes, researchers have conducted various studies to produce Huffman codes or Huffman-like codes using different techniques. Here, the expression Huffman codes means codes with the optimum "prefix free" feature, while the expression Huffman-like means codes with "prefix free" feature close to the optimum.

Moffat and Katajainen (1995) proposed an algorithm that would generate optimal "prefix free" code lengths for large alphabets. With this algorithm, the lengths are produced in two stages. First, the depths of the internal nodes are calculated for the optimal Huffman tree, and then the lengths of the codewords are calculated using these depths. The aim of this algorithm is to save memory ~~by~~ using the "in-place" technique. O (n) spaces are used to generate Huffman codes in O (n) time, provided that the weight array used is sequential.

Moffat and Turpin (1997) proposed a minimum redundancy code technique based on the repetition of symbol probabilities. This algorithm proposed for large alphabets has O (r + r log (n / r)) time and memory space. Here, r defines the number of symbols with different frequency values, and n expresses the number of symbols. The r value should be less than n/2 to consider an improvement in this study. Although this requirement can be met for large alphabets, for small alphabets, r is usually greater than n/2. For example, for the Calgary Corpus, r=72 and n=91 in the" paper2" file, and r=79 and n=81 in the "bib" file.

Other studies have even been done for optimal "prefix-free" codes (Navarro, 2013) (Leeuwen, 1976) (Barbay, 2016). However, these are alphabetical binary tree structures used in data structures and data storage, rather than data compression. In these studies, to find the lengths of Canonical Huffman code words, the optimal Huffman tree is first designed, and then the lengths are determined by going from leaves through the root. Algorithms that calculate code lengths without composing a tree have been proposed. However, these algorithms produce Huffman-like codes.

In 1999, Graham Fyffe introduced near-optimal binary "prefix free" codes (Geldreich, 1999), which he named ~~as~~ Fyffe Codes, and in 2010, Andrew Polar introduced another non-Huffman binary tree, also known as Polar Codes. These algorithms are codes with near-optimal binary "prefix-free" property. It is claimed that the context-based adaptive version of Polar codes is used by Google (Polar, 2010). The build time of the Fyffe and Polar codes is O (kn) because it requires tuning after n steps, as explained in the third chapter.

Dube and Beaudoin (2008) aimed to produce “prefix-free” codes quickly, rather than being optimal, with another algorithm they called “disposal prefix codes”. The construction time of the code tree is O (Δ + n), where F is the frequency of any symbol and Δ= F_max- F_min. In the tree in question, because only one child is allowed at any node, the average bit length per symbol is larger than the optimum compared to its counterparts.

2.3. Previous Works on Text Compression Using Evolutionary Algorithms

This section describes important text compression studies using evolutionary algorithms. These studies were mostly implemented using genetic algorithms (GA) and genetic programing (GP), which are members of evolutionary algorithms.

Üçoluk and Toroslu (1997) used GA to select the best alphabet to use based on letters and syllables before compressing the data. They then, used the chosen alphabet to compress Turkish texts using the standard Huffman algorithm. A Turkish-specific syllable algorithm was used to obtain syllables. In this study, a main alphabet is created that contains the symbols that make up the text to be compressed and all the syllables in the text. With GA, a sub-alphabet that will give the best compression ratio is selected from this alphabet. Sub-alphabets are formed as follows: If a syllable in the main alphabet is to be in the sub-alphabet, it takes the value “1” in binary, otherwise, it takes the value “0”. The syllable with the value “0” is divided into its components, and each component independently joins the sub-alphabet. For example, if the syllable “ab” has the value “0”, the symbols “a” and “b” increase the frequencies of the “a” and “b” characters in the alphabet. Alphabets formed in this way in binary order form individuals. The first 100 individuals ~~are and~~ are randomly generated. The new individuals produced by crossover and mutation are evaluated, with the compression ratio used as the fitness value. The Elitist method is applied in selection. By a predetermined rate, the best individual is replaced by the worst of the previous generation. Upon completion of the process, the text is compressed with Huffman coding using the best alphabet selected. As a result, with including syllables in the alphabet used, a better compression ratio was achieved compared ~~to~~ with the alphabet in which only symbols were used.

Oroumchian et al. (2004) proposed the selection the most appropriate dictionary of 2-gram, 3-gram, 4-gram, and 5-gram words to provide the optimum or near-optimum compression ratio for Persian Web pages and other Internet-based applications. They used GA in their algorithm. Commonly used n-gram words in the language are detected using a training set. n-gram words selected using GA have been replaced with single characters in the user-defined section of the Unicode table. Thus, they reduced the size of the text file by up to 52%. Compression is performed only on the server, and no user decoding is performed, provided that the appropriate Unicode table is preloaded by the user. The method can be applied to any language. However, the success of the method also depends on the training set used to detect n-gram words.

Kuthan and Lánský (2007) were inspired by the approach of Üçoluk and Toroslu. They developed four different universal spelling algorithms that can be used with any language to obtain the syllables. Using a training set of documents written in the language in which the text to be compressed was written, the authors created a master alphabet containing all syllables with a spelling algorithm. The sub-alphabet, which is derived from this main alphabet and provides the best compression ratio, was chosen using GA. Sub-alphabets were created by assigning a value of "1" or "0" to syllables in the main alphabet. The syllables encountered for the first time during compression were separated into their components and coded, later they were added to the syllable table. Later, when this syllable was encountered, they were processed like other syllables. Czech and English texts were compressed with syllable-based LZWL and HuffSyllable algorithms using the best dictionary selected with GA. Compressed texts were compared with Gzip 1.3.5, Bzip2 1.0.3, and Compress 4.2 programs. When the results were evaluated, they showed that the HuffSyllable + GA couple achieved a better compression ratio for Czech texts from 100 bytes to 10 Kbytes and from 100 bytes to 1 Kbytes for English.

Because there is no universal compression algorithm that can compress all data types at the same rate, Kattan and Poli (2008a) proposed a GP system that would divide a file into small pieces and determine the algorithm that would best compress each piece. In this system, the individual (chromosome) is the file to be compressed. The parts are genes. Each gene is assigned a random compression algorithm. Compression algorithms are also selected from a predetermined set of functions. Other individuals are formed by assigning different compression algorithms to the same genes. By applying genetic operations to individuals created in this manner, the individual with the best compression ratio is selected. Then, the same process is repeated by increasing the length of the piece. Thus, by trying all piece lengths, the piece length and algorithm sequence with the best compression ratio are selected. However, the biggest disadvantage of this system is the time cost (1 day for 1 Mb). It also requires header information to determine the length of the segment and which segment is compressed by which algorithm. In the tests, the best results are achieved in one piece for texts and EXE files, since they have a regular data structure. However, a better compression ratio was achieved as the number of parts increased in the archive files comprising different data types. Katan and Poli later made some improvements to their algorithm, which they called GP-ZIP. In their new algorithm, called GP-ZIP*, instead of splitting files into equal-length chunks, they let the algorithm generate chunks of different lengths. Therefore, they rearranged the crossover and mutation operators. Thus, because of editing, they improved performance and compression time. (average 3.73 hours for 1 Mb) (Kattan and Poli, 2008b).

Kattan and Poli took their work described above a step further with their work called GP-ZIP2. In this study, they used two concepts called "separator tree" and "qualifier tree". Separator tree; It divides the file in a training set into blocks of appropriate size using some statistical elements

such as entropy, median, and standard deviation. Each block is compressed with the compression algorithms in the function set to find the most suitable compression algorithm for the block. Using two feature extraction trees, numerical X and Y values to represent the block are found for each block with the assistance of the same statistical elements. Each block is represented in Euclidean space with X and Y values. Blocks are clustered using the K-means algorithm. While the blocks in each cluster have similar properties to each other, they do not have similar properties to the blocks in the other cluster. Each cluster is tagged with the compression algorithm that predominates in the blocks within it. When a file is to be compressed, it is clustered by dividing into blocks similarly, and the proper compression algorithm is determined by comparing each cluster with the clusters in the training set. The task of GP is to find optimal feature extractor trees that produce values to represent blocks in Euclidean space. While GP-ZIP2 is more successful than other programs, especially in heterogeneous archive files, its execution time is slow enough to take several hours (Kattan and Poli, 2011). To overcome this slowness, Katan and Poli developed GP-ZIP3. GP-ZIP3 shares the same structure as GP-ZIP2. However, instead of obtaining the compression ratio by compressing each block one by one with the compression algorithms in the function set, which takes a lot of time in GP-ZIP2, a "decision tree" was used to estimate the compression ratio in GP-ZIP3 (Kattan and Poli, 2009). Thus, the application time was reduced by 5 times. Compression performance is the same as that of GP-ZIP2, as no further improvements have been made despite the speed reduction (Kattan and Poli, 2010).

Boisclair and Wagner (2008) proposed a compression algorithm that allows the use of multi-character symbols with no prior knowledge of the file to be compressed. The GA used in the algorithm uses Huffman trees as individuals. The initial population is trees comprising different sequences of 256 characters of equal length. The algorithm is based on mutation only, not crossover, as the structure of trees causes some sequences to be lost or repeated. With the swap mutation, leaves and internal nodes swap places with each other to form alternate trees. The combine mutation may create multi-character symbols by combining two leaves. GA selects the tree that minimize the compressed file size. As a result, although the proposed algorithm was efficient in files containing human genetic codes, it did not introduce the expected efficiency for executable and text files.

Zaki and Sayed (2009) proposed an algorithm using GP that aims to find a Huffman tree in which the individual forms from the nodes of a tree and minimize the compression ratio. The authors used different operators instead of these operators, because, traditional GP operators cannot guarantee a single code rule (prefix-free) to be assigned to each symbol because of random changes. These are: insertion, bi-level mutation, and changed crossover operators. 1- Insertion operator: used to insert new sub-trees into the existing tree. 2- Bi-level mutation: two nodes (e.g., a and b) are randomly selected with no parent/child relationship between them. Node b is downgraded by one level with its children, while b's parent and a are made sibling. If any, a's previous sibling is upgraded

by one level. Thus, a new tree is formed. 3- Modified crossover: mother and father are not used in this method. A new individual is created by replacing two randomly selected points from different levels of a single individual with their subtrees. In fact, this process is a mutation process, as it is performed by the internal change of a single individual, not using the parent. There is no gene transfer from different individuals. However, the authors preferred to call this process modified crossover. The initial population comprises a single randomly generated tree containing all symbols of the alphabet used. By applying genetic operators to this individual, a new individual is formed and evaluated using the fitness function. The fitness function is the average bit length per symbol. This method allows the use of multiple symbols along with a single symbol. Because the algorithm targets the best Huffman tree, it cannot outperform traditional Huffman codes. However, instead of a single-symbol tree, a multi-symbol tree can be chosen, which can achieve a better compression ratio. The tests carried out also confirm this situation.

K.I.M. Abuzanouneh (2017) proposed a GA algorithm that adopts the multiple Huffman tree approach of characters, syllables, and words to achieve the best compression ratio. In this algorithm, the individual representing an initial Huffman tree comprising 256 characters is represented by a sequence of 1 and 0 bits representing the leaves and internal nodes of the tree. Individuals forming characters, syllables, and words are evaluated separately using the fitness function that minimizes the compression ratio. Among the three best trees selected by GA for coding, the tree with the best compression ratio is selected with a decision tree. To form different individuals, different mutation operators, such as swapping, insertion, and merging, are used with multi-point crossover. Thanks to these mutation operators, syllables and words are produced. The method of selection is elitism. When the test results were examined, it was seen that better results were obtained than the character-based standard Huffman coding, because, it also evaluates syllables and words other than characters. It has also been seen that good results are obtained when compared with other word-based algorithms, such as Huffman and LZ77.

Considering that the most appropriate alphabet to be used for each text file may be different, J. Platos and P. Kromer conducted their research using GA and simulated annealing (SA) algorithms. For this, the text file to be compressed is scanned and a main alphabet covering all symbols up to the desired length (such as 1-gram, 2-gram) is determined. Individuals are created with the values “1” and “0” corresponding to the symbols in this main alphabet. “1” indicates that the corresponding symbol is included in the individual, and “0” indicates that it is not. The fitness function is the compression ratio to be minimized and the size of the alphabet used. The research was implemented by applying GA and SA algorithms with different compression algorithms using the above structure. As a result, 1-gram long (character-based) alphabets for small files (up to 200 KB), 2-gram and 3-gram long alphabets (syllable-based) for medium-sized files (200 KB–5 MB), and word-based

alphabets for relatively larger files are suitable for compression (Platos and Krömer, 2011a) (Platos and Krömer, 2011b) (Platos and Krömer, 2012).

2.4. Previous Works on Adaptive Huffman Encodings

Faller in 1973 and R. Gallager in 1978 independently worked on adaptive Huffman coding (AHC). In his article, Gallager defined the "sibling property" feature, which is an important principle for the formation of a binary Huffman tree. It also describes how to implement the AHC algorithm and how to perform probability updates. However, he did not mention how to start the Huffman tree (Gallager, 1978). Knuth (1985) solved this problem by starting the Huffman tree with a "zero-weight node" in 1985. Thus, the AHC algorithm was put into practice. The algorithm started by Gallager and Faller and completed by Knuth is called Faller-Gallager-Knuth (FGK) algorithm. In his article published in 1987, J.S. Vitter improved the FGK algorithm with new concepts such as "invariant" and "block". Thus, it achieved less height and an optimum average value (average length per symbol). The new algorithm in question is known as the "Vitter algorithm" or "Algorithm Λ". The FGK algorithm and Vitter algorithm will be explained in more detail in the third section.

Wei-Wei Lu and M.P. Gough (1993) proposed an adaptive Huffman algorithm that could encode faster. Two tree structures called "front tree" and "back tree" are used in the proposed algorithm. The front tree is an adaptive Huffman code, but the number of its leaves (codewords) is variable, whereas the back tree is a symmetric or approximately symmetric tree and contains binary code. Each symbol is encoded in only one tree and moves dynamically between two trees according to its frequency of occurrence. Symbols with higher frequencies have a better chance of being encoded in the front tree, whereas those with lower frequencies usually stay in the back tree. The relatively small size of the front tree provides faster adaptation, and thu,s data is encoded more efficiently. The result is faster encoding and better performance.

Tharini and Vanaja Ranjan (2009) proposed an algorithm for wireless sensor networks, which they describe as modified Huffman coding (Modified Huffman Algorithm). In this algorithm, the leaves of the tree do not represent the measurement values themselves, but the set of measurements with the same frequency. For this reason, transmitted binary codes comprise two parts, a prefix and a suffix. The prefix indicates the corresponding leaf in the tree, and the suffix indicates the order of the measure in the cluster represented by that node. The transmitted measurement values are the difference between the actual and previous measurement values. The measurement sets in the leaves are comprise these difference values. In this way, they achieved a lower-level tree structure and a better compression ratio.

Klein et al. (2020) proposed an adaptive Huffman coding algorithm, which they developed from a different perspective and called forward coding. In the known adaptive Huffman algorithms (Vitter and FGK), when a character is encountered, coding is performed according to the previously

formed Huffman tree. This is backward coding. Klein et al., encode according to the tree that may be in the future (forward coding). For this purpose, the characters and frequencies are determined by scanning the compressed file. On the basis of these values, a Huffman tree is created. This tree is the last tree formed in the static and adaptive Huffman algorithms. When a character is encountered during the compression process, it is encoded according to this tree. The frequency of the character is reduced by one. The tree is updated by decreasing the number of nodes on the path from the leaf to the root to which the character belongs. If the frequency of a character is zero, that character is removed from the tree and the tree is rebuilt. Removing characters with zero frequency from the tree is a feature that makes the algorithm different. Thus, since the tree will get shorter and shorter, it will produce a shorter code. Forward coding has been shown to have better compression performance than the static Huffman and Vitter algorithms. However, the starting tree for the decoder also needs to be transmitted (Klein et al., 2021). To use forward coding, the source and its statistical information must be known beforehand. Klein et al. also developed a different version of forward coding. In the version called hybrid encoding, the characters and frequencies are determined by scanning the file to be compressed beforehand. However, as in forward coding, the tree is not created. Similar to the Vitter algorithm, the NYT (Not Yet Transmitted) character is used to define the characters encountered for the first time. Initially, there is a single-node tree containing the NYT character, which has frequencies equal to the number of distinct characters in the message to be compressed. When the first character is received, NYT is encoded first, followed by the character's ASCII code and frequency. Elias coding is used to encode the frequency. Each time the NYT is encoded, its frequency decreases by one. After encoding, the tree is updated by adding characters. When a character in the tree is received, it is encoded according to the current tree. After its frequency is coded, its frequency is decreased by one. The tree has been updated. Characters with a frequency of zero are removed from the tree. Thus, as the number of characters decreases, shorter encoding can be provided (Klein et al., 2019). However, to use both advanced coding and hybrid coding, the source must be known beforehand. Where the source is not known beforehand, they cannot be used, for example, for coding instantaneously measured values.

3. MATERIALS AND METHODS

3.1. Materials

3.1.1. Introduction

The materials section is organized into three main headings.” Concepts of Data Compression” aims to summarize fundamentals of information theory and data compression. Of course, this summary covers only the concepts closely related with this thesis.

For canonical Huffman-like coding, which is the subject of this thesis, an algorithm that calculates the lengths of the code words algebraically (ACHC) produces near-optimum canonical Huffman codes. In **“**Canonical Huffman and Canonical Huffman-Like Coding” section**,** Canonical Huffman coding, Fyffe and Polar codes that produce Huffman-like codes, and FGK and Vitter versions of adaptive Huffman coding are explained because of their relevance to the thesis topic. Because, the ACHC algorithm was optimized using evolution strategies, evolution strategies are also described.

3.1.2. Information Theory

In our daily lives, we transmit a lot of information to each other without knowing what information means. Expressions such as the red color of a book, the magnificent beauty of the evening sun, and the melody of a song carry information and we can transmit this information in written, visual, and audio form. There are various explanations in dictionaries for the definition of information. However, in terms of data compression, it is not important what the definition of information is, but the amount of information and how it can be measured. Claude Elwood Shannon examined the measurement of information, which is necessary for the transmission and storage of information, and the principles he determined were generalized under the name of "Information Theory" (Rioul, 2021).

Shannon defined a numerical value that he called self-information. Let be “an event A” that forms the result set of a random experiment. If the probability of event A is P(A), self-information about A is given by Equation (3.1).

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A) \tag{3.1}$$

Self-information i (A) defines the amount of information that “event A” carries. If b=2, bit is the unit of the amount of information. If b=10, then the unit is a digit (or Hartley). After this point, b will be taken as 2 because it will work in binary order. If we know “event A” for certain, that means P(A)=1. Then, i (A)= $-\log_2 1 = 0$. This means that since event A is already known, it has no information to

give us. As P(A) approaches zero, the information about the event decreases and the amount of information carried by the event increases at the same rate. This situation can be explained with the following example: let be a coin with heads on both sides. When coin is tossed, it will come up head because there is no other possibility. The probability of getting a head is P (H)= 1, and it has no information to give. Therefore, i (H) =0. However, if one side of the money is a tail, it cannot be known which one will come. If the probability of getting a head is P(H)=1/8, then i(H) = 3 bits from Equation (3.1). The probability of getting a tail is also P(T)=7/8 and i(T)=0.193 bits. These results indicate that event P(H) carries more information than the event P(T). To put it more clearly, 2^3=8 different messages can be carried with 3 bits, while $2^{0,193} \cong 1$ messages can be transmitted using 0.193 bits.

Another feature of the mathematical definition of information is that the information obtained from the occurrence of two different events is the sum of the information from the individual occurrence of the events. Let X and Y be two independent events. From Equation (3.1), the following equation can be written

.

$$i(XY) = \log_2 \frac{1}{P(XY)}$$

Because X and Y are independent, the equation P(A) = P(X) P(Y) can be written, and self-information is as follows.

$$i(XY) = \log_2 \frac{1}{P(X)P(Y)} = \log_2 \frac{1}{P(X)} + \log_2 \frac{1}{P(Y)} = \text{i(X)} + \text{i(Y)}$$

If there is a set of events A, such as A= {X, Y}, the mean eigenvalue of set A is defined as follows:

$$H = \sum P(A)i(A) = -\sum P(A) \log_2 P(A)$$

Let this definition apply to an M message that contains information. Let S be an n-element alphabet that composes the message. If the frequency of occurrence of each symbol in the S alphabet that makes up the M message is p(s_i), where 1 ≤ i ≤ n, the average self-information of the M message is defined as follows:

$$\text{H(M)} = -\sum_{i=1}^{n} p(s_i) \log_2 p(s_i) \qquad (3.2)$$

Shannon called this average self-information "Entropy". Entropy, in terms of data compression area, refers to the minimum size at which an information source can be compressed without losing the information it carries. The unit is bits per symbol (b/S). For example, let an M="kasasakatasa" message be. The alphabet that makes up the M message is S= {a, s, k, t} and their probabilities are P(A)= {6/12, 3/12, 2/12, 1/12}, respectively. The entropy of the M message is:

$$H(M)= -(\frac{6}{12}\log_2\frac{6}{12} + \frac{3}{12}\log_2\frac{3}{12} + \frac{2}{12}\log_2\frac{2}{12} + \frac{1}{12}\log_2\frac{1}{12}) = 1{,}73 \text{ b/S}$$

Since the M Message comprises a four-element alphabet, it can be expressed with 2x12=24 bits using 2 bits per symbol. However, entropy indicates that the same message can be expressed without loss of information, with a minimum of 1.73 bits per symbol, for 1.73x12=20.76 bits (Sayood, 2018).

Equation (3.2) defines the entropy created by independent events. This type of entropy is called zero-order entropy. If the probability of a symbol s_i from a source depends on the symbol s_{i-1}, the entropy of this type of source is called first-order entropy. Likewise, if the probability of any symbol from the source depends on the previous two symbols, it is called second-order entropy. If it depends on the previous three symbols, it is called third-order entropy. With these definitions, algorithms based on higher order entropies are designed, and thus smaller compressed sizes are achieved.

3.1.3. Concepts Used in Data Compression

While discussing any subject, it is necessary to know the related concepts to better understand that subject. In this section, the concepts used in data compression, especially within this thesis, are explained below.

Source and Alphabet

Any source is the starting point of information comprising symbols. The source can be a text file or an image. Within the content of this thesis, the terms “message” or “file” will be used in the sense of source. Sources can be named independent and dependent sources according to the relationship of the generated symbols with each other. If the symbols coming out of the source have no relationship with each other, or if the output of any symbol is not dependent on the previous symbol, this type of source is an independent source. If any symbol from the source is evaluated according to the symbol before it, or if the probability value of the symbol is evaluated as the probability of B after E (P (B/E)), then this source is the dependent source. In this thesis, compressed sources (Messages, files) will be handled as independent sources. The set of different symbols produced from a source is called the *Alphabet*. Each s_i element that makes up the S alphabet is called

a symbol or character, where $1 \leq i \leq n$. The number of characters that make up the S alphabet will be denoted by n.

Codes and Encoders

When a message or a file is created, a binary number called a code or codeword is assigned to each character used. Their length and value depend on the code table used. For example, in the ASCII code table, each code is 8 bits long and has 256 characters. Such codes are called "fixed length codes". If codes are displayed in different lengths, such codes are called "variable length codes". Table 1.1.a. fixed length and Table 1.1.b. are examples of variable-length code. Encoders are algorithms that convert the code of one symbol into another code. Decoders are algorithms that reverse the encoded symbols to obtain the original information. Encoders are divided into three parts according to how they compress data: static encoders, semi-static encoders, and adaptive (dynamic) encoders.

a. Static Encoders: Static data compression compresses all data using the same model, regardless of the content and statistical information of the data files. For example, the Golomb encoding algorithm is a static encoder because it applies the same model to all data files (Kalaivani and Tharini, 2020).

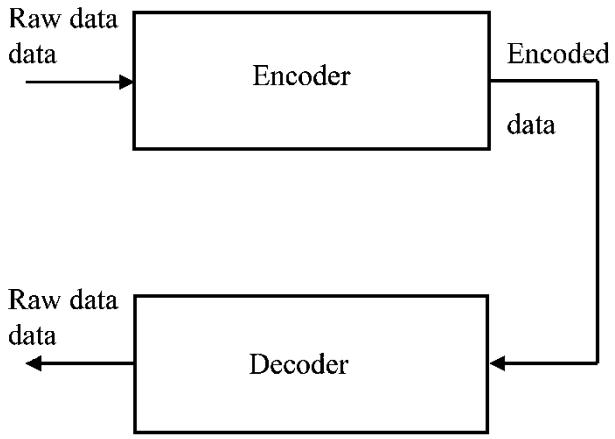


Figure 3.1. Encoder and Decoder

b. Semi-static Encoders: Semi-static encoders create a separate model for each data file based on the probability distribution. Then, the data are compressed according to the created model (according to the statistical information). They are also known as two-pass algorithms because, they pass over the data once for model building and once for compression. (Marjai, Lehotay-Kery ve Kiss, 2022). Traditional Huffman coding is an example of this (Moffat ve Petri, 2020).

c. Adaptive Encoders: Semi-static encoders cannot be used in instantaneous situations where data are constantly changing. Because in this case, the probability distribution is unpredictable. In addition, two passes over the data increase the compression time. As a solution to these negative situations, adaptive (dynamic) compression algorithms have emerged. In adaptive applications, the

character to be compressed is compressed according to the probability distribution of the preceding characters. After the character is compressed, the probability distribution table is updated. Adaptive algorithms are one-pass algorithms because the data compression process is performed in one pass. Adaptive Huffman coding (AHC) is an example of this type (Sayood, 2018).

Compression Performance

Some measurement methods are used to evaluate the efficiency of the compression algorithm.

Compression ratio (CR): the ratio of the compressed file size to the original file size.

$$CR = (Size\ of\ compressed\ file)/(Size\ of\ uncompressed\ file) \quad (3.3)$$

According to this formula, for example, the equation "CR=0.7" express that the compressed data corresponds to 70% of the original file. This means that the size of the original file has been reduced by 30%. The smaller the CR than 1, the better the compression. The CR>1 status express enlargement.

Compression Factor (CF): The compression factor is the inverse of the compression ratio.

$$CF = (Size\ of\ uncompressed\ file)/(Size\ of\ compressed\ file) \quad (3.4)$$

Here, the greater the CF, the better the compression. A value less than 1 indicates expansion.

Saving: It expresses the acquisition obtained by the compression process as a percentage.

$$Saving = 100\ x\ (1 - CR) \quad (3.5)$$

In this criterion, for example, saving=60 express that 60% savings are achieved with the applied compression algorithm.

Average Bit Length

The efficiency of variable-length codes is measured by the average bit length per symbol (A_v). For any message A_v, if the code length of the i^h symbol is l_i and its probability is p_i, where 1 ≤ i ≤ n and n is the number of symbols used, the average bit length per symbol is defined by Equation (3.6).

$$A_v = \sum_{i=1}^{n} p_i\, l_i \quad (3.6)$$

The difference between A_v and entropy defines the residual information (Redundancy, R).

$$R = A_v - H = (\sum_{i=1}^{n} p_i\, l_i) - (-\sum_{i=1}^{n} p(s_i) \log_2 p(s_i))$$

A case of R > 0 indicates that there is still information to be compressed. The ideal situation is R=0.

Kraft-MacMillan Inequality

A binary tree designed to represent variable codes is defined by the Kraft-MacMillan inequality, where each internal node has two children and the codes corresponding to the leaves in that tree are not prefixed by any other code in the same tree (prefix-free feature). The Kraft-MacMillan (KM) inequality is defined by Equation (3.7), where n is the number of symbols in the alphabet, the code length or the number of bits in the code is l, and $1 \leq i \leq n$ (Solomon, 2007).

$$KM = \sum_{i=1}^{n} 2^{-l_i} \leq 1 \tag{3.7}$$

The condition KM < 1 ensures the prefix property but indicates that there is an internal node with missing children in the tree. The case KM =1 prescribes that all internal nodes have two children and is the ideal case.

Model

The compression process is divided into two parts: modeling and coding. The model provides information about symbols to be coded to the encoder. For example, consider the message consisting of the following sequence of numbers.

M = 17, 18, 19, 18, 16, 17, 19, 18, 20, 22, 24, 26, 28

When this sequence is examined, it can be seen that there are numbers near each other. Therefore, by keeping the first number, we can get smaller numbers when we take the difference between the next number and the previous number.

M_d = 17, 1, 1, -1, -2, 1, 2, -1, 2, 2, 2, 2, 2

The M message can be modeled as $M_d = m_i - m_{i-1}$, where $m \in M,\ i = 1,2 \ldots$ Thus, the M message can be compressed using fewer bits with the differential model. The decoder can regenerate

the entire original message by adding the second number to the first number in the M_d difference message, then adding the third number to the resulting number, and so on.

The probability distribution table, which contains the probability values of the symbols used in a message, is also a model for the encoder. For example, in the message M= "kasasakatasa" the alphabet is A= {a, s, k, t} and the probabilities are P(A)= {0.5,0.25,0.167,0.08}. Here, P(A) is the model for the encoder. Figure 3.2. describes the model. "Input stream" means the data stream from the source, and "output stream" or "bit stream" means the encoded data stream coming out of the encoder.

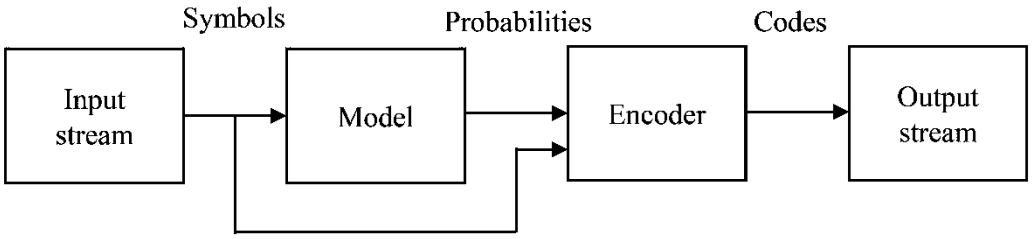


Figure 3.2. Modeling of probabilistic encoders

Taxonomy

Since the Shannon-Fano encoder, which was first designed as a probability encoder, various data compression algorithms have emerged for different purposes and applications. In Section 3.2.2, the encoders were divided into three parts: static encoders, semi-static encoders, and adaptive encoders according to the way they compress the data.

A general classification of the data compression algorithms is illustrated in figure 3.3. Data compression algorithms (encoders) are divided into two main parts: lossless compression and lossy compression. If a message can be retrieved with the decoder in a lossless manner after it has been compressed, it is lossless compression. Encoders that compress text and programs use lossless compression algorithms. In lossy compression algorithms, an amount of information is lost after the data is compressed. Therefore, the message cannot be fully retrieved. Assume that N is the compressed version of a message M. Let O be the decoded version of N. If M = O, then compression is lossless, otherwise, compression is lossy.

Lossy compression algorithms take advantage of the weaknesses of the eye and ear and are used in image, video, and audio compression. The human eye and ear cannot detect high frequencies and tiny, subtle changes. When components that cannot be detected by the eye and ear are removed, a visible change in sound and image cannot be perceived by the human. Extraction is performed by applying frequency transformations such as discrete cosine transform (DCT) or discrete wavelet transform (DWT) to image and audio. After this conversion, the original file cannot be fully retrieved.

Lossless encoders are also divided into two parts: variable-length encoders (prefix encoding) and fixed-length encoders (dictionary encoding).

Dictionary encoders create a dictionary (table) from symbols and sequences of symbols used to process the message. They give index numbers of symbols and of symbol sequences in the dictionary to the output. Compression is achieved by representing long symbol strings with shorter dictionary indices. .

Variable-length encoders are also divided into two types: entropy encoders and integer encoders. Integer encoders are used to encode numbers. They generate variable-length codes.

Entropy (probability, statistical) encoders encode data according to the frequency (probability) of the symbols used in the message to be compressed. The shortest code is assigned to the most probable symbol, and the longest code is assigned to the least likely symbol.

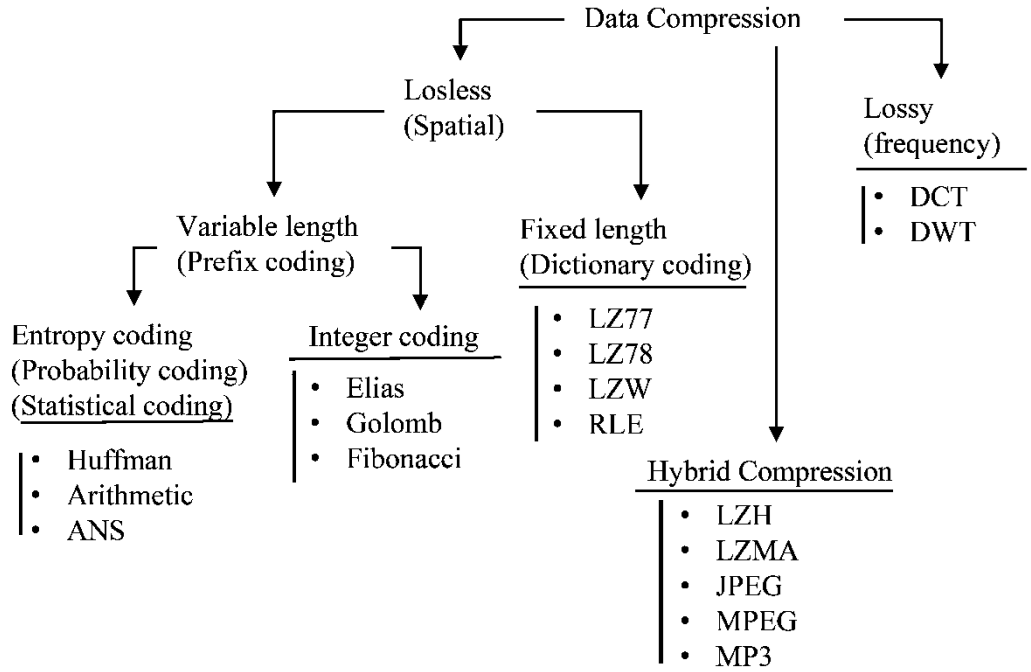


Figure 3.3. Classification of the Data Compression Algorithms

Hybrid encoders use multiple encoders together. For example, after the LZMA dictionary has been encoded, it re-encodes the output with an entropy encoder. In image, audio, and video (JPEG, MPEG, MP3, etc.) compressions, RLE and/or an entropy encoder are used after frequency (lossy) conversion.

3.1.4. Canonical Huffman and Canonical Huffman-like Coding

Canonical Huffman Coding

Huffman coding is based on the principle of assigning the shortest codeword to the most probable symbol, and falls into the class of "prefix-free" codes with variable length. The average bit

length per symbol gained by Huffman Encoding is the closest average value to entropy after the Arithmetic Encoder.

Given an ordered frequency table (or probability distribution) of a message, Huffman begins the coding process by combining the two symbols with the lowest weight (Figure 3.4.). Thus, the newly formed symbol is added to the table. The table is rearranged. Again, the two lowest-weighted symbols in the table are combined. This process continues until all symbols in the table are combined into one symbol. At the end of the process, the Huffman tree is formed. The codewords associated with the symbols in the leaves of the Huffman tree are found by going from the root to the leaves.

Canonical Huffman codes are a subset of Huffman codes. It has a "numeric sequence feature". The numeric sequence feature means that binary values denoting codewords of the same length are consecutive. For example; like "000", "001", "010", "011". Canonical codes have some advantages: Because the length values are sequential, the size of the "header" that needs to be sent to the decoder with the compressed file is significantly reduced. It is sufficient to send the length values to the decoder together with the alphabet used. This increases the encoding and decoding speed and reduces the amount of memory used (Ranjan ve Kumar 2023).

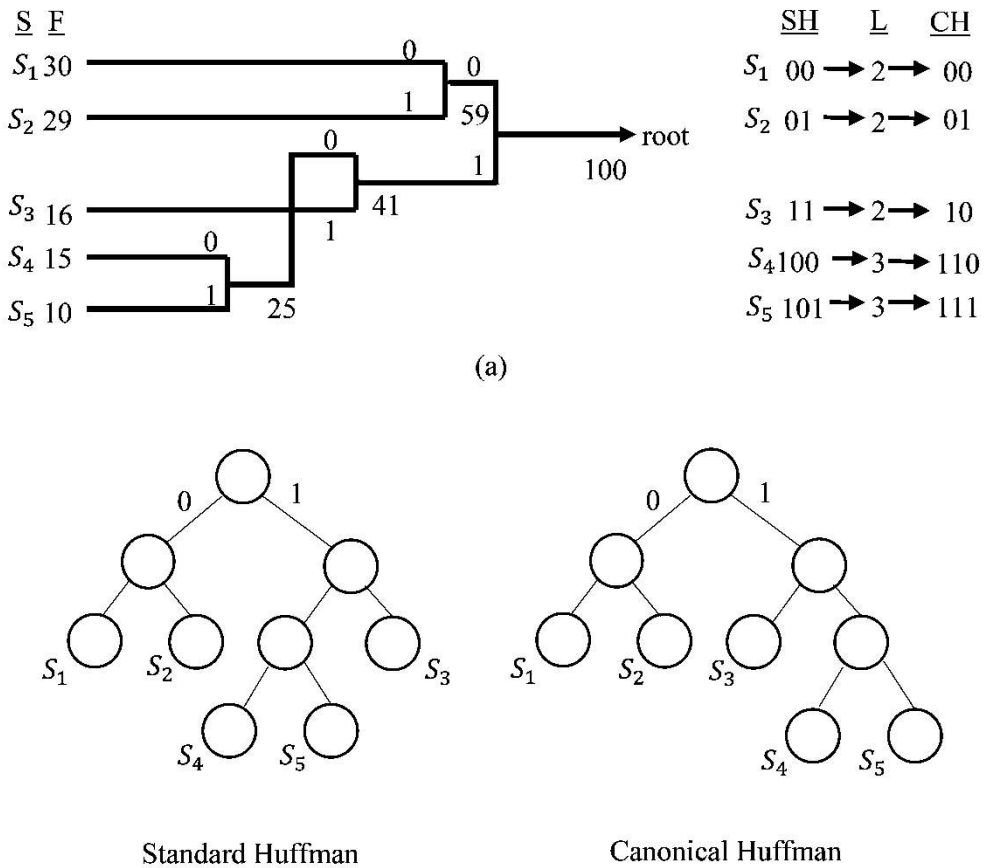


Figure 3.4. a. Determination of codewords and lengths in the Huffman tree b. Binary tree representation
S: symbols, F: frequencies, SH: standard Huffman, CH: canonical Huffman

Figure 3.4. illustrates the production of Huffman codes and Canonical Huffman codes. Canonical Huffman codewords are generated from the Huffman tree according to the length of the codewords. First, the Huffman tree is designed according to the symbol weights. The length (depth or number of bits) of the codewords for each symbol is determined by traversing the tree from the root to the leaves representing the symbols. The codeword corresponding to the smallest length is always bit "0". If the length is greater than 1, "0" is added to the right by the length value (shift left). For example, if the length value is 2, the codeword is "00". The next code word is found by adding "1" in binary to the previous codeword. If the length of the specified codeword is shorter than the corresponding length value, "0" is added to the right of the codeword by the difference (shift left). For example, if the length value is 3 and the codeword determined by adding "1" to the previous codeword is "11", the bit value "0" is added to the right of the codeword to make the code length 3 and the new codeword becomes "110". The table in Figure 3.4.a contains the canonical codes for an example message with five symbols.

The efficiency of the Huffman codes is measured by the average bit length per symbol (A_v) given by Equation (3.6) (Hidayat, Zakaria ve Pee, 2023). Time complexity of Huffman coding; if the weight array is not ordered, it is O (nlogn). The space complexity is O (5n) (Moffat and Katajainen, 1995). The code example of the Huffman algorithm is given in Appendix B.

Fyffe Codes

The purpose of Fyffe Codes is to quickly estimate the codeword lengths generated by the Huffman algorithm. Suppose S is an alphabet of n symbols for any message, $S = \{s_1, s_2, \dots, s_n\}$. The probabilities of S are P=$\{p_1, p_2, \dots, p_n\}$. Also, let the codeword set be $C = \{c_1, c_2, \dots, c_n\}$ where $c_i \in \{0,1\}$. l_i is the length of c_i, that is, the number of bits it contains, where L is the set of lengths and $l_i \in$ L. The length of any codeword is given in Equation (3.8) (Rioul, 2021).

$$l_i = -\log(p_i) \tag{3.8}$$

However, since l_i must be an integer, Equation (3.8) can be written as:

$$\lfloor -\log(p_i) \rfloor \leq l_i \leq \lceil -\log(p_i) \rceil$$

The main idea is to initially set the length of each codeword to $\lceil -\log(p_i) \rceil$ and gradually decrease the length value by one degree until KM=1 (Equation (3.7)). Here Equation (3.7) indicates that the binary code tree representing the codewords is a complete tree, meaning that the inner nodes have two children. This also means that the codewords are unique, meaning they have the "prefix-free" feature.

A R (residual value) variable is defined to determine which l_i value to be reduced:

$$R = 1 - \sum_{i=1}^{n} 2^{-l_i} \quad (3.9)$$

R<0 says that in the code tree, there is a node with missing children, and R>0 defines a node with various children. When R=0, a complete set of codewords is created. 2^{-l_i} and R are compared to decide which length value to reduce. If 2^{-l_i}> R, no action is taken. If $2^{-l_i} \leq$ R, l_i is decreased by 1. We can explain the process with a simple example. Let our symbol set (alphabet) be S= {A, B, C, D} and probability distribution P= {0.6, 0.25, 0.1, 0.05}. For each step; l_i, 2^{-l_i} values corresponding to s_i values and R value are calculated. The process steps are shown in Table 3.1. In step 1, each rounded up value of l_i is found and R is calculated according to Equation (3.9). Starting with the

Table 3. 1. Fyffe Coding

S	P	L	2^{-l_i}
A	0.6	1	0.5
B	0.25	2	0.25
C	0.1	4	0.0625
D	0.05	5	0.03125
R= 0.15625			

Step 1

L	2^{-l_i}
1	0.5
2	0.25
3	0.125
5	0.03125
R= 0.09375	

Step 2

L	2^{-l_i}
1	0.5
2	0.25
3	0.125
4	0.0625
R= 0.0625	

Step 3

L	2^{-l_i}
1	0.5
2	0.25
3	0.125
3	0.125
R= 0	

Step 4

most frequent weighted symbol, each 2^{-l_i} term is sequentially compared with R. The values 2^{-l_1} and 2^{-l_2} are greater than R=0.15625. 2^{-l_3} is less than R, l_3, is reduced by 1. In step 2, the value 2^{-l_4} is less than the updated R and l_4 is reduced by 1. In the next step, because the value 2^{-l_4} is equal to R, it is reduced by 1 again. Now that R=0, the process is completed. The final L values are the lengths of the codewords that are prefix-free. The next step is to derive the code words in canonical form using these lengths.

Polar Codes

Polar codes are variable length and "prefix-free" codes used for data compression, such as the Huffman encoder. The calculation of the Polar codes is as simple as that of the Fyffe codes. Symbol frequencies are used instead of probabilities to determine the lengths. The polar method works as follows: Let the alphabet and its frequencies be S= $s_1, s_2, \dots, s_n$ and F= $f_1, f_2, \dots, f_n$. Let the sum of the frequencies be T= $\sum_{i=1}^{n} f_i$, and let l_i be the length of any codeword c_i from the set of codewords C= $c_1, c_2, \dots, c_n$. All symbols are already ordered by frequency. First, the frequencies are summed, and the total T_1 is rounded up to the nearest power of 2. Each frequency value is rounded down to the nearest power of 2. Frequency values written as powers of two are multiplied by two (doubling) by going from top to bottom in the sorted list. The doubling continues until the total T_n of the frequencies is greater than or equal to T_1. If $T_1 = T_n$, doubling is stopped. If T_n is greater than

T_1, all frequency values are sequentially divided by two, starting from the first row, until $T_n = T_1$. When $T_n = T_1$ the process is completed. The length is then calculated by applying the formula:

$$l_i = \log T_n - \log F_i \tag{3.10}$$

F_i is the i^{th} frequency value in the last step. Let us explain the generation of Polar codes with the example in Section 3.4.2. Our alphabet is the set S= {A, B, C, D} and their frequencies F= {60, 25, 10, 5}. The process steps are shown in Table 3.2. First, the frequencies are rounded down to the nearest power of 2. The first sum of frequencies is rounded up to the nearest power of 2 ($T_1 = 128$). The rounded frequency values are then doubled and their sum (T_n) is calculated. Since T_n<T_1, the doubling continues. In the next step, T_n is equal to 240 and doubling is stopped. Since T_n is greater than T_1, the final frequency values are sequentially divided by two until T_n=T_1. If needed, the division process can be repeated starting from the beginning. When T_n=T_1, the lengths of the codewords are calculated according to Equation (3.10). The next step is to find the code words using their lengths.

In reality, Polar codes and Fyffe codes are not very different algorithms. Only difference: Polar codes round the logarithm of weights down, whereas Fyffe codes round up. Before the setting stage, when the R value is calculated from the lengths, it will be seen that it is R>0 for Fyffe codes and R<0 for Polar codes. In both, the goal is to reset the R value.

Table 3.2. Polar Codes

S	F	Rounding	doubling 1	doubling 2	tuning 1	lengths
A	60	32	64	128	64	1
B	25	16	32	64	32	2
C	10	8	16	32	16	3
D	5	4	8	16	16	3
Total	100 (T)	128 (T_1)	120 (T_n)	240 (T_n)	128 (T_n)	

3.1.5. Adaptive Huffman Coding

Faller-Gallager-Kunt (FGK) Algorithm

For a tree formed by encoding each s_x character in a message M, comprising n symbols (characters) different from each other, to be a Huffman tree, the sibling property must be satisfied.

For 1< x < n, each s_x character in the message M has a weight w_x, which corresponds to the frequency of occurrence in the message. Each node (parent) with two children creates internal nodes, and their weights are the sum of their children's weights. Here, leaves and internal nodes are sorted and numbered in ascending order of weight. In sorting, every $2j - 1^{th}$ and $2j^{th}$ element is said to be

a sibling. Here, j is defined as $1 \leq j \leq n-1$. The sibling feature is shown in Figure 3.5. Here nodes 1 and 2, 3 and 4, 5 and 6 are siblings.

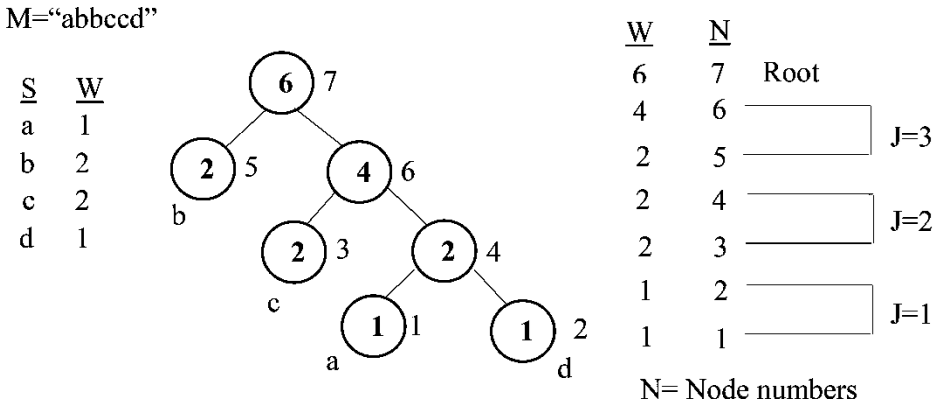
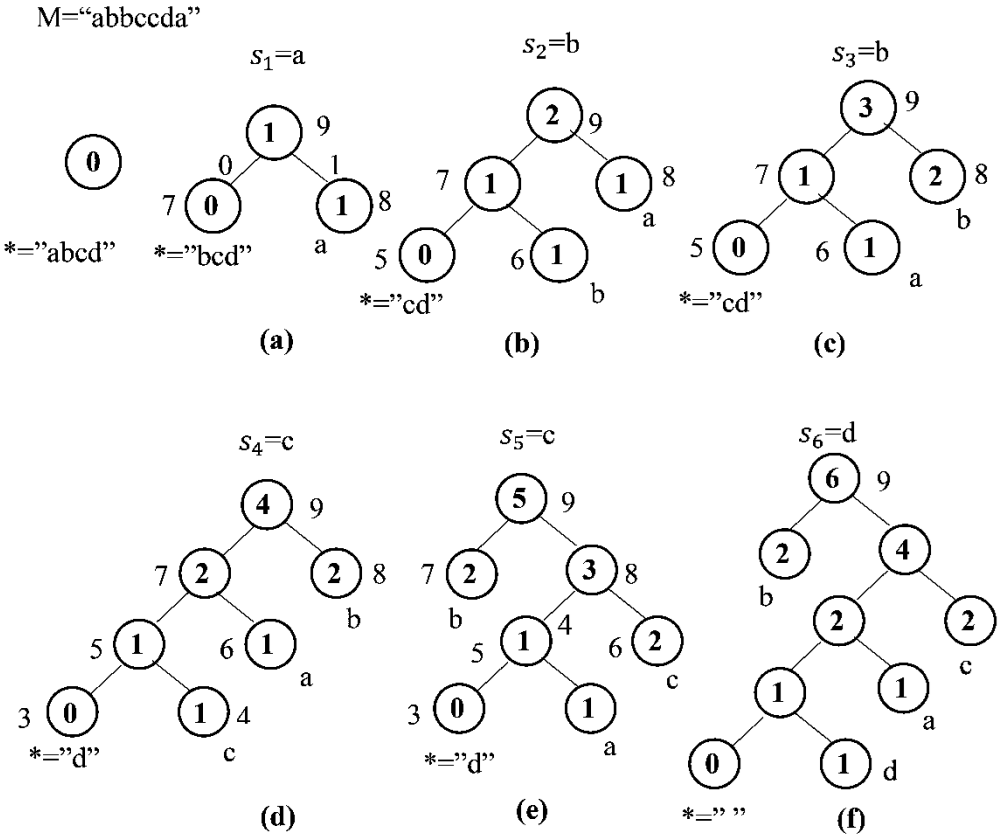


Figure 3.5. Sibling property

In the FGK algorithm, the tree is initially initialized with a weighted 0 node. This node contains unused characters from the alphabet used. This node, which is shown as NYT (Not Yet Transmitted) in the literature, will be represented by the * symbol throughout the thesis. Two children of the * node are created when a new character arrives for processing. The new left child (leaf) created becomes the new zero-weighted * node. The one on the right is the new character to be added to the tree. If the character to be encoded exists in the tree, it is encoded according to the existing tree. Otherwise, the code of the * node in the current tree is broadcast first, followed by the code of the new character, which is also known by the decoder (for example, the ASCII code). The tree is then updated. Updating is performed by increasing the weights of the nodes on the path from the character to the root. However, before the increase is made, the node whose weight will be increased is replaced with the highest numbered node that is the same as its current weight. The update process is performed by increasing the weights of the nodes on the path from the character to the root. This change continues until the root. Finally, the update is completed by increasing the weight values. Displacement can be left to right or bottom to top. Figure 3.6. illustrates the encoding of the M= "abbccda" message using the FGK algorithm. The tree is initialized with a zero-weighted * node. Because the first character "a" is not in the first tree, the binary code describing the "0" corresponding to the * symbol followed by the "a" character is output. The tree is updated with the addition of the character "a" (Figure 3.6.a.). Since the next character "b" is not in the updated tree, first the "0" code corresponding to the * symbol and then the binary code corresponding to the "b" character are sent. The tree has been updated (Figure 3.6.b.). The next character is "b" again. The corresponding binary code "01" is issued because it is contained in the current tree. The tree has been updated (Figure 3.6.c.). The next character "c" does not exist in the current tree. Therefore, the "00" code corresponding to the * symbol is output, followed by the binary code that defines the "c" character. The tree has been updated (Figure 3.6.d.). Because the next "c" character is in the current tree, the

corresponding “001” code is issued. The tree is updated. At this point, because the current weight value of the character “c” whose weight value will increase is 1, it replaces the node with the highest numbered node with the same weight and containing the character “a” (nodes 4 and 6). Because the value of parent node 7 will increase, it will be replaced by node 8 with the same weight. Finally, the update is completed by increasing the value (Figure 3.6.e.). Because the next character “d” is not available, the “100” code corresponding to the * character is output, followed by the binary code identifying the “d” character. The tree has been updated (Figure 3.6.f.). Since the last character “a” is in the tree, the corresponding code “101” is added to the output and the coding is completed. Thus, the encoding (compression) of the message is completed.



Output: 0 “a” – 0 ”b” – 01 -00 “c” – 001 – 100 “d” – 101 => abcd + 15 bits

Figure 3.6. FGK Algorithm

The decoding process also consists of the same stages. Since the first bit in the received encoded message is the “0” character, the next bits of a certain length are decoded as the “a” character from the related table, for example, from the ASCII table. The tree in Figure 3.5.a. is created. Because the next bit “0” is the “*” character according to the resulting tree, the fixed-length bits following it are resolved as “b”. The tree in Figure 3.6.b. is formed by updating the tree again. Because the next bit “0” corresponds to the internal node on the tree, the next bit “1” is passed and the character “b” is reached. The process continues in this manner, and the entire message is received. Because the

update is along the leaf-to-root path, the time required to encode a character in the FGK algorithm is O(l), where l is the distance from the leaf-to-root path. The total memory requirement is $12n\ log_2 n + 2n\ log_2 t$ bits, where t is the total number of characters encoded (Knuth, 1985).

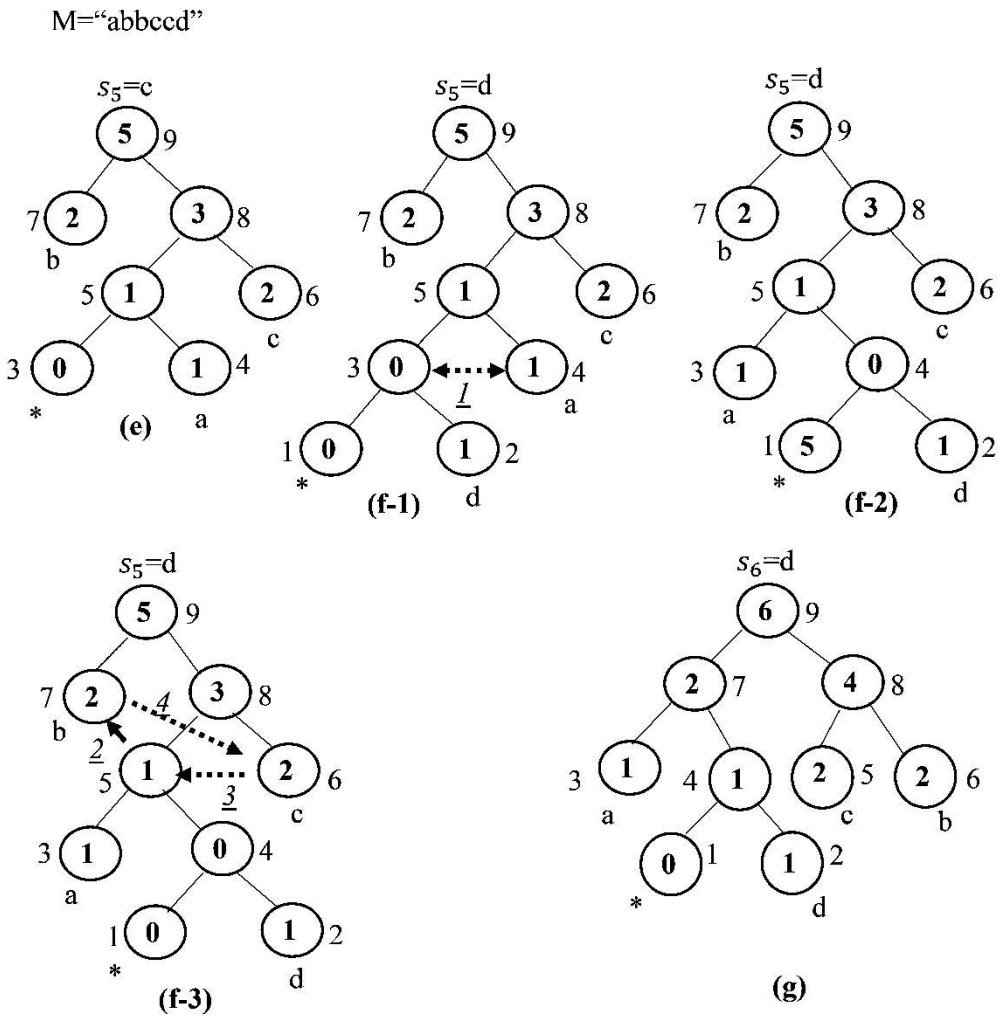
Vitter Algorithm

Vitter (1987) improved the FGK algorithm by introducing two new features. As a result, minimum $\sum_j l_j$ and $max_j \{l_j\}$ values were obtained, where l is the length from the leaf to the root (bits of the corresponding code or depth of the tree). The characteristics defined by Vitter are:

a. Implicit numbering: All nodes on the tree are sorted in descending order of their weight, starting from the root. All nodes are numbered, with the highest number belonging to the root. Numbers increase from bottom to top and from left to right at the same level. The implicit numbering corresponds to the visual representation of the tree in the Vitter algorithm. The numbering performed in the FGK algorithm is called explicit numbering and corresponds to the physical locations of the nodes. Unlike implicit numbering, in explicit numbering, the weight order does not change, but the numbering order can change (Vitter,1987).

b. Invariant property: When all nodes are numbered, the leaves of weight w for each weight of w precede the nodes of weight w. W-weight leaves form one block, and w-weight nodes form a separate block. The node or leaf with the largest number in each block is called the leader. The invariant feature is provided as follows: During the update, for the s leaf with a weight w, first, the nodes with the weight w on the right of it are shifted to the left. Then, the s leaf is placed in the formed space and its weight is made w+1. If a node with a weight w is to be updated, first the leaves with a weight w+1 on its right are shifted to the left. Then, the node is placed in the created space and its weight is set to w+1. Because to these two features, the Vitter algorithm can provide a minimum tree structure and an optimum average bit length per character. In Figure 3.7., the application of the Vitter algorithm to the M=" abbccda" message is shown. Some stages of the process are the same as those of the FGK algorithm, and some stages are different owing to the "invariant feature". Therefore, the update processing after Figure 3.6.e is illustrated to show the difference. Figure 3.7.a step is the same as FGK. When the new "b" character is received, the * leaf is split into two in the tree shown in Figure 3.6.a. Because the new left leaf will increase by 1 from the zero weight of the new * node, it is replaced by the node 8 as per the principle of invariance. The new tree formed is symmetrical to the tree in Figure 3.6.b. The tree created when the second "b" character is taken is the same as that in Figure 3.6.c. When the "c" character is added to this tree, the value of node 5 will increase, so it will be replaced by node 6. Then, node 7, which is the parent of nodes 5 and 6, is replaced by node 8 as its weight increases. As a result, the symmetry of the tree shown in Figure 3.6.d occurs. It should be noted that, while the nodes are relocated, the content is moved, and the node numbers are not moved. The tree formed when the second "c" character is received is the same as that in Figure 3.6.e as shown in Figure 3.7.e. The update after the next "d" character is received is shown in Figure 3.7.f.

When the “d” character is added to the tree, the weight of node 3 will increase, and it will be replaced by leaf number 4. Because the weight of node 5, which is the parent, will also increase by 1, it replaces node 7. Leaves 7 and 6 form a block of 2 weights and the leader is leaf number 7.



Output: 0 “a” – 0 ”b” – 11 -00 “c” – 101 – 100 “d” – 00 => abcd + 14 bits

Figure 3.7. Vitter Algorithm

According to the principle of invariance, replacement is performed with the leader. Node 5 is shifted and number 6 is exchanged with node 5. Then, node 7 is in the place of number 6. It should be noted that in all displacements, only the contents are replaced. After the displacements are completed, the weights are increased, and the update is completed. Thus, Figure 3.7.f is obtained. As seen, a tree with less depth is formed compared to the tree in Figure 3.6.f. This is an advantage of the Vitter algorithm over the FGK algorithm. Since the last character “a” is present in the tree, the corresponding code “00” is added to the output, and the process is completed. The total output length is 1 bit shorter than that of the FGK algorithm.

The time complexity required to encode a character in the Vitter algorithm is O(l), where l is the tree depth (or the number of bits of the code). The space complexity is 16n $\log_2 n$+15n+2n

$\log_2 t$ bits, where t is the total number of characters encoded (Vitter, 1987). A code example of the Vitter algorithm is given in Appendix C.

3.1.6. Evolution Strategies

Evolution strategies (ESs) are used to find the optimum or near-optimum solutions in a search space for a given problem. ESs use mutation and elitist selection as search operators to solve real-world problems. Chromosomes are represented by strategy parameters and pairs of individuals. Let X_n represent an n-dimensional individual and SP be the strategy parameter. Here, a chromosome is denoted by the expression $C_n = (X_n, SP_n)$. The SP strategy parameter is the parameter that mutates the individual to which it belongs to. SP evolves to have an optimal value in each generation. This feature is the most important feature of ESs. Different adaptation methods for SP have been suggested by researchers. A simple method can be represented by Equation (3.11):

$$SP = \sigma Z \tag{3.11}$$

σ is the mutation step size and controls the strength of the mutation. Z is a normally distributed random vector that can be expressed as $Z \sim N(0, \boldsymbol{I_n})$. I_n is the n-order unit matrix. Z can be defined using different probability functions according to the problem. The mutation is applied to SP first. The individual is then mutated through SP. The stages of mutation are shown by Equation (3.12) and Equation (3.13). Let us consider X(g) as an individual of the g^{th} generation. Here, t ≥ 0. $f(X(g))$ is the fitness function value of X in the g^{th} generation.

$$SP(g+1)=\text{mutate}(SP(g)) \tag{3.12}$$

$$X(t+1) = \begin{cases} X(g) + SP(g+1) & f\big(X(g) + SP(g+1)\big) \le f\big(X(g)\big) \\ X(g) & otherwise \end{cases} \tag{3.13}$$

A symbolic notation is used to represent the ESs structure. This notation is usually expressed by the expression (μ+ λ) – ES or (μ, λ) – ES. Where μ represents the number of parents, λ represents the number of children produced from μ parents. (μ+ λ) – ES notation shows that the best μ individuals are selected from among μ parents and λ children (elitism). The range of values for μ and λ is defined as that: $1 \le \mu \le \lambda < \infty$. (μ, λ) – ES notation expresses that the best μ individuals are selected from among only λ children. The range of μ and λ values is $1 \le \mu < \lambda < \infty$ (Günter, 2000). Some types of ES can be represented by different notations. For example, (μ+ λ) – SA-ES indicates that SP is self-adaptive (Auger, 2005) and (μ+ λ)–CMA-ES show the adaptation of SP with the covariance matrix (Auger and Hansen, 2005).

3.1.7. Calgary Corpus

The Calgary Corpus is a collection of binary and text files commonly used to evaluate data compression algorithms. Created in 1987 by Ian Witten, Tim Bell, and John Cleary of the University of Calgary (The Calgary Corpus, 2022). It is still widely used today. There are 14 various files in Corpus, including text and binary files. The files in the Calgary Corpus are listed in table 3.3.

Table 3.3. Calgary Corpus

File	Abbrev	Category	Size
bib	bib	Bibliography (refer format)	111.261
book1	book1	Fiction book	768.771
book2	book2	Non-fiction book (troff format)	610.856
geo	geo	Geophysical data	102.400
news	news	USENET batch file	377.109
obj1	obj1	Object code for VAX	21.504
obj2	obj2	Object code for Apple Mac	246.814
paper1	paper1	Technical paper	53.161
paper2	paper2	Technical paper	82.199
pic	pic	Black and white fax picture	513.216
progc	progc	Source code in "C"	39.611
progl	progl	Source code in LISP	71.646
progp	progp	Source code in PASCAL	49.379
trans	trans	Transcript of terminal session	93.695

3.2. Methods

3.2.1. Introduction

The three proposed algorithms are introduced in this chapter. Algebraic Canonical Huffman Coding (ACHC), which calculates the code lengths in one stage through direct calculation, is the first algorithm proposed in this thesis. The second proposed algorithm is ES_ACHC which optimizes the ACHC algorithm with the Evolutionary Strategies algorithm. Finally, the adaptive application of the ACHC algorithm (A_ACHC) was also discussed within the scope of this thesis.

3.2.2. Algebraic Canonical Huffman Coding

The algorithm proposed in this thesis has a top-down design similar to Fyffe and Polar codes, unlike the traditional Huffman algorithm. However, the average bit length per symbol is closer to the optimum value than the average per symbol length of the Fyffe and Polar codes. The goal of this algorithm is to obtain optimal or near-optimal “prefix free” code lengths simply and quickly. The entire process is completed in $\Theta(n)$ time, provided that the weight array used is sequential. The

memory requirement is Θ(n) words. Here, n is the number of symbols. From this point on, the proposed algorithm will be referred to as ACHC (Algebraic Canonical Huffman Coding).

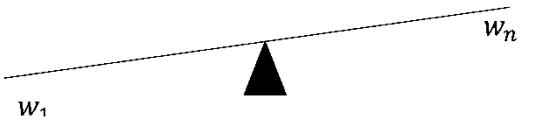
Suppose we have a seesaw. Let us put the weights we have on this seesaw in order from one end to the other. Naturally, the heaviest part will come down (Figure 3.8.a). Our goal is to bring this seesaw to equilibrium or the closest to equilibrium. To do this, we place counterweights, heavier at the lightest end of the seesaw and lighter at the heaviest end. The weights in our seesaw represent the frequencies or probabilities of symbols in a message. The counterweights used to balance these weights are the lengths of the codewords (or bit numbers) that correspond to the symbols used. Therefore, length values should be arranged on the seesaw so that the heaviest part corresponds to the shortest length and the lightest part corresponds to the longest value (Figure 3.8.b). Our problem is to determine the lengths of the codewords necessary to bring the seesaw to or closest to equilibrium and match it to the weights. Shannon's information theory and the observations give us the relationship between weights and lengths (Figure 3.8.c). Let us have an alphabet and a probability distribution of symbols: S= $\{s_1, s_2, \dots, s_n\}$, P= $\{p_1, p_2, \dots, p_n\}$. Here, n is the number of symbols and P is the weights in the seesaw. Shannon stated that the length of the codeword associated with the symbol s_i with the probability p_i will be calculated using Equation (3.8) (Rioul, 2021). However, as a length value, the formula is rounded to the nearest integer, since l_i cannot be a fractional number. On the other hand, for values of $p_i > 0.7$, Equation (3.8) produces values of $l_i \leq 0.49$. In this case, the value of l_i is taken as 1 directly thus l_i is not zero, as it will be rounded to the nearest integer. Thus, the following formula is obtained, where R is the rounding function to the nearest integer:

$$l_i = \begin{cases} 1 & if\ p_i \geq 0.7 \\ R(-\log(p_i)) & otherwise \end{cases} \tag{3.14}$$

In reality, the length array containing the code lengths corresponds to a binary prefix-free tree. Equation (3.7) must be satisfied to obtain a complete tree structure. Therefore, we will need some parameters.

During the process, as p_i gets smaller, an error occurs in the calculation of l_i. Therefore, scaling becomes necessary. Scaling is performed using two parameters: e_i and k_i. k_i is the cumulative probability values calculated up to the moment of operation: $k_i = \sum_{x=1}^{i-1} p_x$. e_i is the sum of the remaining probabilities: $e_i = 1 - k_i$. Thus, in its scaled form, Equation (3.14) becomes the following formula:

$$l_i = \begin{cases} 1 & if\ p_i \geq 0.7 \\ R(\log(e_i / p_i)) & otherwise \end{cases} \tag{3.15}$$

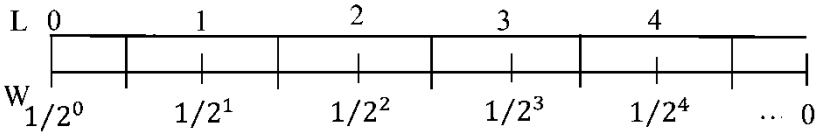


$$w_1 \geq w_2 \geq \cdots \geq w_n$$

(a)

$$l_1 \leq l_2 \leq \cdots \leq l_n$$

(b)



(c)

Figure 3.8. Combining weights and lengths

Considering the parameters on an imaginary tree, the node to which e_i belongs is the reference node. l_i is the distance from the reference node to the ith symbol. The distance from the root to the i^{th} symbol is u_i. Let us call the length from the root to the reference node d_j. If d_j=0, the reference node is the root. If d_j≠0, the length of the i^{th} codeword is given by the formula $u_i = d_j + l_i$, for $1 \leq j \leq n-1$. u_i is the actual length of the codeword corresponding to the i^{th} symbol. Let N be the number of nodes and leaves at distance d_j. The number of leaves and nodes at distance d_j from any reference node is given by the formula $N_j = 2^{l_i}$. The position of the symbol s_i at a distance d_j is defined by the equation , $t_m = (2^{u_i - u_{i-1}} * t_{m-1})$ - 1 with $0 \leq m \leq N_j - 1$. The values of the e_i and d_j parameters change according to the previous value of the t_m parameter:

$$e_i = \begin{cases} e_i = 1 - k_{i-1} & if\ t_{m-1} \leq \dfrac{N_j}{2} \\ e_i = e_{i-1} & otherwise \end{cases}$$

$$d_j = \begin{cases} d_j = d_{j-1} + 1 & if\ t_{m-1} \leq \dfrac{N_j}{2} \\ d_j = d_{j-1} & otherwise \end{cases}$$

For s_n, Equation (3.15) equals zero since $e_n = 1 - k_{n-1}$ and $e_n = p_n$. Therefore, u_n can be taken directly as d_j. All parameters are shown in Figure 3.9. with the help of an imaginary tree with "prefix free" feature.

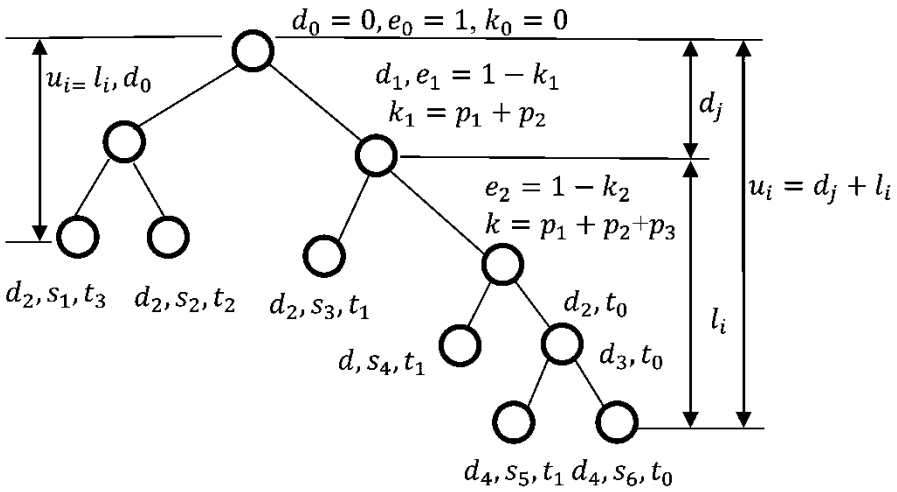


Figure 3.9. Parameters on an imaginary tree with "prefix free"

Thus, all the parameters used to calculate the length of any codeword have been defined. Once the lengths are calculated, the codewords are also easily determined in canonical form. For example, if U= (1,2,3,3) then the codewords would be C= (0, 10,110,111).

ACHC Pseudo-Code

The algorithm that calculates the lengths of the codewords associated with the symbols whose weights are given is given below. The input data are the weight array and the total weight value. If the weight is given as a probability distribution, the total weight value is 1. After the i^{th} length is calculated, it is written in the i^{th} position of the weight array. Therefore, an array of length n is sufficient to calculate the lengths. The code example of the ACHC algorithm written in C is given in Appendix A. A code example of the standard Huffman algorithm is given in Appendix B.

```
ACHC ()
Input: ordered weight array W(0:n); total weight tw
Parameters: e =1, k =0, d=0, t=1, N=1, l=0, i=1; p=0;
Begin
Do while i< n            // number of symbols in the alphabet
{   p=W_i / tw;
    if (p / e ≥ 0,7 and i < n) then l=1
    else l=round(log_2 (e/p)) ;
    if i=n then l=0;
// Since i^th probability is no longer needed, the     calculated length is stored in
//  W_i
    // ^ is the sign of the power
    t= [2^ (W_i – W_(i−1))*t]-1;
    N=2 ^ l;
    k=k + p;
    if t ≤ N/2 then
        e=1 - k, d= d + 1;
    i++;
}
End
Output: Length array W
```

Algorithm 3.1. ACHC Algorithm

Implementation of the ACHC algorithm

Implementing the algorithm to the M="abbccda" message is shown in Figure 3.10. In the sequence of probability given sequentially, the lengths are calculated starting from the most weighted probability downward. Therefore, the algorithm works top-down, not bottom-up as in the standard Huffman algorithm. The output of the algorithm is the code length. Canonical codes are found from these lengths, and compression is then performed using these codes. Becouse the algorithm calculates the length of a single symbol in one loop, the calculation of the lengths of n symbols occurs in the n loop.

s_i	W	p_i	k	e_i	d_j	l_i	t_m	N_j	**U**	Code
			0	1	0	0	1	1	0	
b	2	0,3	0	1	0	2	3	4	**2**	00
c	2	0,3	0,3	1	0	2	2	4	**2**	01
a	2	0,3	0,6	0,4	1	1	1	2	**2**	10
d	1	0,1	0,9	0,1	2	0	0	1	**2**	11
Total	7	1								

M="abbccda" => output="10-00-00-01-01-11-10" $= \sum_{x=1}^{i-1} p_x$

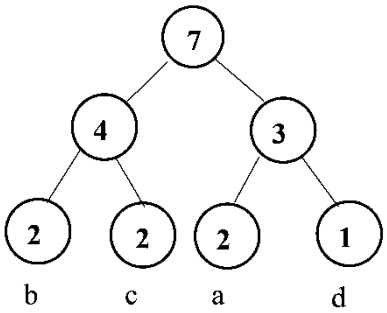


Figure 3.10. Implementation of the ACHC algorithm

Time Complexity and Space Complexity

The time complexity of an algorithm is the number of times of the statements in that algorithm. The time complexity (O(n)) is obtained by evaluating the asymptotic approximation of the equation (T(n)) consisting of the number of times the statements execute and their cost (the execution time of the statement as a constant number). In the asymptotic approximation, the highest-order variable of the equation is taken and the other variables, coefficients and constants are discarded. This is because as n goes to infinity, components other than the highest-order variable are neglected (Cormen, Leiserson, Rivest, & Stein, 2009).

The cost and number of runs of the expressions in the ACHC algorithm are in Table 3.4. The variable n is the number of symbols in the alphabet used and is the input of the algorithm.

Table 3.4. Time complexity of ACHC algorithm

Statements	**Cost**	**Times**
Input: ordered weight array W(0:n); total weight tw	c_1	1
Parameters: e =1, k =0, d=0, t=1, N=1, l=0, i=1; p=0;	c_2	1
Do while i< n	c_3	n+1
{		
p=W_i / tw	c_4	n
If (p / e ≥ 0,7 and i < n)	c_5	n
then l=1 else l=round($\log_2$ (e/p)) ;	c_6	n
If i=n then l=0;	c_7	n
t= [2^ (W_i – W_{i-1})*t]-1;	c_8	n
N=2 ^ l;	c_9	n
k=k + p;	c_{10}	n
If t ≤ N/2 then e=1 - k, d= d + 1;	c_{11}	n
i++;	c_{12}	n
}		
End		

From Table 3.4, we can write the T function as follows:

$$T(n) = c_1 * 1 + c_2 * 1 + c_3 * (n+1) + c_4 * n + c_5 * n + c_6 * n + c_7 * n + c_8 * n + c_9 * n + c_{10} * n + c_{11} * n + c_{12} * n$$

If organized:

$$T(n) = (c_1 + c_2) + c_3 + n * (c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12})$$

$$A = c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12} \qquad B = c_1 + c_2 + c_3$$

$$T(n) = An + B => O(n)$$

Since the ACHC algorithm runs in n steps in each case, the time complexity is Θ(n).

For the determination of code lengths, an integer array of length n+1 is sufficient to be used as a weight array and length array. The calculated ith length value is recorded on the ith weight. Each element of the W array must hold the t total number of characters in the processed message, because t is greater than the *l* length. Therefore, the required bits number is $nlog_2(t)$ bits (if n >> 1, 1 can be neglected) for n + 1 symbols (with the initial value for U).

The memory consumed in the standard Huffman algorithm is 5n words. 2n words are used in n leaves for symbol weights and parent information. Another 2n words are for the weights and parent information used in the remaining n-1 nodes. In addition, the priority queue used for the construction of the Huffman tree requires n words of memory. Thus, a total of 5n words of memory are used (Moffat,2020). In this method, n word memory units are used for weight calculation. For an accurate comparison, log(t) = 1 word can also be used for the weight calculation for ACHC. In this

case, the amount of memory consumed by the ACHC algorithm is n words. The memory savings of the ACHC algorithm compared with the standard Huffman algorithm is calculated as 80% from the formula savings=1- n/5n. This shows that using the ACHC algorithm saves 80% of memory compared with the classical Huffman algorithm. The ACHC algorithm has shorter code lines than the Huffman algorithm. The code example of the ACHC algorithm written in C is in Appendix A. A code example of the standard Huffman algorithm is given in Appendix B.

3.2.3. Optimizing the ACHC Algorithm with Evolution strategies (ES_ACHC)

Near-optimal lengths of Canonical Huffman code words are found by the ACHC algorithm. This sequence of lengths forms the ancestor in the ES algorithm. The ES algorithm used can be described with (1 + λ) – CSP-ES notation. Here, the number 1 defines a single parent (μ = 1), which is the ancestor. The ancestor is replicated by copying as many as λ and these copies mutate. As a result, λ children are produced. The best individual is selected from among the current ancestor and children. The selected best individual becomes the ancestor of the next generation. CSP (Constant Strategy Parameter) means that the strategy parameter (SP) has not evolved. SP is constant because it does not evolve. Because, in the equation shown by Equation (3.4), the random vector elements σ=1 and Z only take the values +1 and -1; Z $\{z_1, z_2, \dots, z_n\}$, $z \in \{-1,0,1\}$. The algorithm used can be considered as a sexless reproduction model (mitosis): A single ancestor reproduces by copying itself. Then, the copies mutate. The best individual survives and becomes the ancestor of the next generation.

The process continues until the termination criterion is met. The flow chart of the algorithm is shown in Figure 3.11. The output of the algorithm is an array of lengths of Canonical Huffman code words.

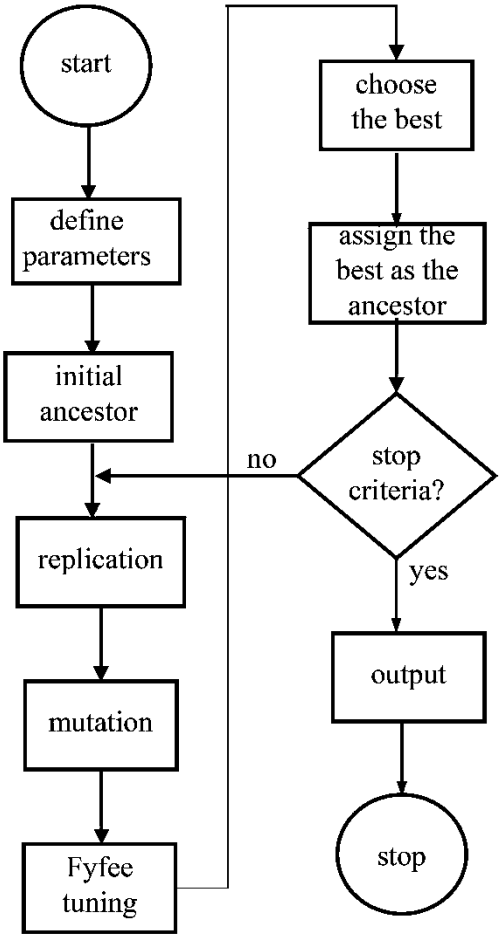


Figure 3.11. Flow chart of the ES_ACHC algorithm

Individual

In an evolutionary process, chromosomes are the initial step of EA. In the ESs algorithm, chromosomes are in the form of pairs composed of strategy parameters and individuals defined by the equation $C(g) = (X(g), SP(g))$. In this expression, $C(g)$ is the chromosome, $X(g)$ is the individual and $SP(g)$ is the strategy parameters and "g" is the number of generations. Individuals are possible solutions to given problems. Here, individuals represent arrays of length (or bit numbers) of codewords in a Huffman tree. Any gene (element) in an individual is the length of the codeword corresponding to a leaf in the Huffman tree.

The ES_ACHC algorithm uses a single parent, called the ancestor. The length array calculated by ACHC is used as the first ancestor. This sequence is the sequence with the closest length to the optimum. The ancestor is mutated after replicating as many as λ. Figure 3.12, illustrates an example of an individual and the corresponding Huffman tree.

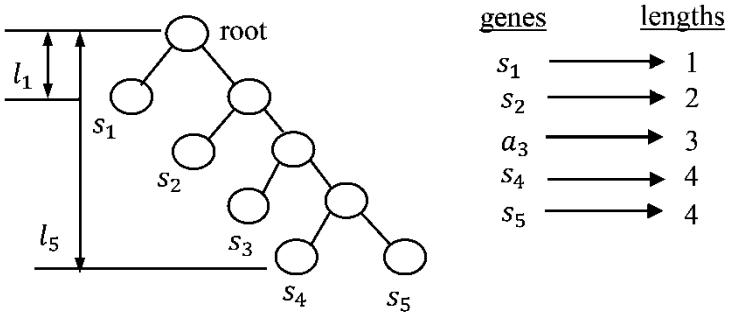


Figure 3.12. An example of an individual with five elements

Mutation

Mutations are random changes in one or more genes. After the ancestor replicates as many as λ copies, all copies are mutated. Thus, λ different children are produced. The mutation is applied by adding or subtracting 1 from the SP value of each child. Here, $SP = \sigma Z$ and $\sigma = 1$, Z is a random vector. The Z vector is defined as Z ={$z_1, z_2, \dots, z_n$} with $z \in \{-1,0,1\}$. Here, the mutation is: $L(g+1) = L(g) + Z(g) = \{l_1 + z_1, l_2 + z_2, l_3 + z_3, \dots, l_n + z_n\}$, where n is the number of symbols in a message. Initially, z is equal to zero, and at the mutation point, it takes the value +1 or -1 with 50% probability. The mutation has two predetermined parameters: the number of mutation points (MP) and the mutation rate (MR). MPs are selected in two ways: 1. Transition points 2. Midpoints. Transition points are points where at least one neighbor of any element in the length array has a value different from itself. Midpoints are points where both neighbors have the same value (Table 3.5.). Mutation points are randomly selected from the transition points and midpoints. The number of MR * λ of individuals that mutates from the transition point, and the number of (1-MR) * λ from the midpoints. Differences in code lengths close to the average bit length per symbol are mostly at transition points. Therefore, MR is chosen close to 1. Designing the mutation in this way according to the characteristics of the problem can be defined as "target-directed mutation". Mutated individuals may be structurally impaired. The length array representing a Huffman tree must satisfy the Kraft-MacMillan (KM) inequality (Equation 3.7) to provide the prefix-free property. For this reason, all individuals are reviewed using the "Fyffe tuning" algorithm in case of deterioration after mutation, and the corrupted individuals are amended. Undoubtedly, the children who suffered from structural disorders (who did not provide the KM inequality) could have been eliminated, and continued with the healthy ones. However, in this case, the diversity decreases, the evolution time takes a long time and reaching the best value can not be guaranteed.

Table 3.5. Mutation points

P	optimal	Near optimal	midpoints	transition points
0,260	2	2		
0,173	2	2	X	
0,173	3	2		X
0,130	3	4		X
0,086	3	4		X
0,043	5	5		X
0,043	5	5	X	
0,043	5	5	X	
0,043	5	5		
Av	2,913	2,956		

Fyffe Tuning

After the mutation operation, the tree structure corresponding to the length array may be corrupted. Then, the KM inequality (Equation 3.7) cannot be achieved. The correction process, called Fyffe tuning, applies to the individual deteriorated its structure. Fyffe tuning takes its name from the Fyffe coding developed by Graham Fyffe in 1997 and is described in Section 3.1.4.2. Fyffe coding is used to produce near-optimal prefix-free codes (Oral and Aşşık, 2019). In this study, Fyffe coding was used to mend the individual whose structure had deteriorated with some minor changes. Thus, the KM inequality is restored. The Fyffe adjustment steps are shown in Table 3.5. l_i is the corrupted length array element, the equations are $KM= \sum_{i=1}^{n} 2^{-L_i}$ and the residual value is R=1-KM. The purpose of tuning Fyffe is to set the R value to zero.

Table 3.6. Fyffe tuning

l_i	2^{-l_i}	l_i	2^{-l_i}	l_i	2^{-l_i}	l_i	2^{-l_i}	l_i	2^{-l_i}
1	0,5	2	0,25	2	0,25	2	0,25	2	0,25
1	0,5	1	0,5	1	0,5	1	0,5	1	0,5
4	0,0625	4	0,063	3	0,125	3	0,125	3	0,125
5	0,0313	5	0,031	5	0,031	4	0,0625	3	0,125
KM=1,0938		KM=0,844		KM=0,906		KM=0,9375		KM=1	
R=-0,094		R=0,156		R=0,094		R=0,0625		R=0	
Step 1		Step 2		Step 3		Step 4		Step 5	

Starting from the first element, the length values are increased by 1 or decreased by 1 depending on the R value. If R<0, the length value is increased by 1. If R>0, it is decreased by 1. If R=0, the process is complete. Once R has a positive value, it is not allowed to take a negative value. A negative R value is the difference between Fyffe encoding and Fyffe tuning. In the first step of the example in Table 3.6., R is negative. Therefore, the initial value is increased by 1 (step 2) and R becomes positive. Then, the second value (step 3) is reduced by 1, R becomes negative. Here, the operation is undone, and the third element is then decreased by 1 (step 4). The third element does not change because R will be negative again. In the fifth step, the fourth element is reduced by 1, and R=0. Thus, the operation is completed.

Selection and Fitness Value

The average number of bits per symbol A_v (Equation 3.6) is used for the fitness function. Here, the goal is to find the smallest A_v value. In the selection stage, deterministic selection is used to find the best individual. For this, the fitness value of each individual is calculated first. Then, the individual with the smallest fitness value is selected from among the ancestor and children. The chosen individual is the ancestor of the next generation. The process continues until the predetermined stop criterion is satisfied. After enough loops, the code lengths corresponding to the best codewords are obtained. Canonical codewords are then produced from these lengths, and Huffman coding is completed using these codewords.

Time and Space Complexity

The ES_ACHC algorithm consists of five subroutines. The time complexity of the algorithm is the sum of the time complexities of these subroutines. The time complexities of the subroutines is as follows. Variable n is the number of symbols in the alphabet and is the input of the algorithm. Because the number of offspring is 5*n in length n, the space complexity of the algorithm is also $O(n^2)$ bytes.

Table 3.7. Time complexity of replicate subroutine

Statements	Cost	Times
Parameters	c_1	1
For i<prnt_nmbr // prnt_nmbr=5*n	c_2	5*n+1
Memcopy (Copy 5*n offspring from an ancestor of length n)	c_3	5*n*n
End		

$$T_r(n) = c_1 * 1 + c_2 * 5 * n + c_2 + c_3 * 5 * n * n$$

$$T_r(n) = 5 * c_3 * n^2 + 5 * c_2 * n + c_1 + c_2$$

$$A = 5 * c_3 \qquad B = 5 * c_2 \qquad C = c_1 + c_2$$

$$T_r(n) = A * n^2 + B * n + C \Rightarrow O(n^2)$$

Table 3.8. Time complexity of Mutation subroutine

Statements	Cost	Times
Input parameters	c_1	1
Set parameters	c_2	1
For i < prnt_nmbr // prnt_nmbr=5*n	c_3	5*n + 1
Generate random numbers(rndm(t))	c_4	5*n*1
For t < Mp // Mp=Mutation points constant number	c_5	5*n*Mp + 1
Generate random number (rndmR)	c_6	5*n*Mp*1
If rndmR <= Mp	c_7	5*n*Mp*1
Mutation at transition points	c_8	5*n*Mp*1
Else		
Mutation at midpoints	c_9	5*n*Mp*1
End		

$$T_m(n) = c_1 * 1 + c_2 * 1 + c_3 * 5 * n + c_3 + c_4 * 5 * n * 1 + c_5 * n * M_p + c_5 + c_6 * n * M_p * 1 + c_7 * 5 * M_p * 1 + c_8 * 5 * n * M_p * 1 + c_9 * 5 * n * M_p * 1$$

$$T_m(n) = c_1 + c_2 + 5 * n * (c_3 + c_4 + c_5) + 5 * M_p * n * (c_6 + c_7 + c_8 + c_9)$$

$$T_m(n) = c_1 + c_2 + 5 * n * (c_3 + c_4 + c_5 + M_p * (c_6 + c_7 + c_8 + c_9))$$

$$A = c_3 + c_4 + c_5 + M_p * (c_6 + c_7 + c_8 + c_9), M_p \text{ is a constant } B = c_1 + c_2 + c_3 + c_5$$

$$T_m(n) = A * 5 * n + B \Rightarrow O(n)$$

Table.3.9. Time complexity of Fyffe tuning subroutine

Statements	Cost	Times
Parameters	c_1	1
For i < prnt_nmbr // prnt_nmbr=5*n	c_2	5*n + 1
For t < n	c_3	5*n*(n+1)
Calculate kraft parameter	c_4	5*n*n
Res=1-kraft	c_5	5*n*1
While Res !=0.0	c_6	5*n*(n+1)
Repair birey(i) // Since each individual (birey) is n length, n steps are processed	c_7	5*n*n
i = i + 1	c_8	5*n*n
End		

$$T_f(n) = c_1 * 1 + c_2 * 5 * n + c_2 * 1 + 5 * n * c_3 * (n+1) + 5 * n * c_4 * n + 5 * n * c_5 * 1 + 5 * n * c_6 * (n+1) + 5 * n * c_7 * n + 5 * n * c_8 * n$$

$$T_f(n) = 5 * c_3 * n^2 + 5 * c_4 * n^2 + 5 * c_6 * n^2 + 5 * c_8 * n^2 + 5 * c_2 * n + 5 * c_3 * n + 5 * c_5 * n + 5 * c_6 * n + 5 * c_7 * n + c_1 + c_2$$

$$T_f(n) = 5 * n^2 * (c_3 + c_4 + c_6 + c_8) + 5 * n * (c_2 + c_3 + c_5 + c_6 + c_7) + c_1 + c_2$$

$$A = 5 * (c_3 + c_4 + c_6 + c_8) \qquad B = 5 * c_2 + c_3 + c_5 + c_6 + c_7 \qquad C = c_1 + c_2$$

$$T_f(n) = A * n^2 + B * n + C \implies O(n^2)$$

Table.3.10. Time complexity of Fitness subroutine

Statements	**Cost**	**Times**
Parameters	c_1	1
For i < prnt_nmbr // prnt_nmbr=5*n	c_2	5*n + 1
For t < n	c_3	5*n*(n + 1)
Calculate fitness for every individual	c_4	5*n*n
For i < prnt_nmbr // prnt_nmbr=5*n	c_5	5*n + 1
Choose the fittest	c_6	5*n*1
End		

$$T_{fit}(n) = c_1 * 1 + c_2 * 5 * n + c_2 + c_3 * 5 * n^2 + c_3 * 5 * n + c_4 * 5 * n^2 + c_5 * 5 * n + c_5 + c_6 * 5 * n$$

$$T_{fit}(n) = c_3 * 5 * n^2 + c_4 * 5 * n^2 + c_2 * 5 * n + c_3 * 5 * n + c_5 * 5 * n + c_6 * 5 * n + c_1 + c_2 + c_5$$

$$T_{fit}(n) = 5 * n^2 * (c_3 + c_4) + 5 * n * (c_2 + c_3 + c_5 + c_6) + c_1 + c_2 + c_5$$

$$A = 5 * (c_3 + c_4) \qquad B = 5 * (c_2 + c_3 + c_5 + c_6) \qquad C = c_1 + c_2 + c_5$$

$$T_{fit}(n) = A * n^2 + B * n + C \implies O(n^2)$$

Table.3.11. Time complexity of Acceptance subroutine

Statements	**Cost**	**Times**
Parameters	c_1	1
For i < n	c_2	n + 1
Elite(I)= newelite(i)	c_3	n*1
End		

$$T_a(n) = c_1 + c_2 * n + c_2 + c_3 * n$$

$$T_a(n) = n * (c_2 + c_3) + c_1 + c_2$$

$$A = c_2 + c_3 \qquad B = c_1 + c_2$$

$$T_a(n) = A * n + B \implies O(n)$$

Table.3.12. Time complexity of Es_ACHC Algorithm

Statements	**Cost**	**Times**
Parameters	c_1	1
Ata =ACHC()	c_2	O(n)
While i < L //L= a constant	c_3	L+1
Replicate ()	c_4	L*O(n^2)
Mutation ()	c_5	L* O(n)
Fyffe_Tuning ()	c_6	L * O(n^2)
Fitness ()	c_7	L* O(n^2)
Acceptence ()	c_8	L* O(n)
i = 1+1	c_9	L*1
End		

If the time complexities of the subroutines are substituted into the ES_ACHC algorithm, the equation T(n) is obtained:

$$T(n) = c_1 + c_2 * O(n) + c_3 * L + c_3 + c_4 * L * O(n^2) + c_5 * L * O(n) + c_6 * L * O(n) + c_7 * L * O(n^2) + c_8 * L * O(n) + c_9 * L$$

$$T(n) = O(n^2) * (c_4 * L + c_6 * L + c_7 * L) + O(n) * (c_2 + c_5 * L + c_8 * L) + c_1 + c_3 * L + c_3 + c_9 * L$$

$$A = c_4 * L + c_6 * L + c_7 * L \qquad B = c_2 + c_5 * L + c_8 * L \qquad C = c_1 + c_3 * L + c_3 + c_9 * L$$

If the asymptotic approximation is applied to the resulting T(n) equation, time complexity is obtained.

$$T(n) = A * O(n^2) + B * O(n) + C \Rightarrow O(n^2)$$

3.2.4. Adaptive Algebraic Canonical Huffman Coding

By applying the adaptive compression method to the ACHC algorithm, the Adaptive Algebraic Canonical Huffman Coding (A_ACHC) algorithm is obtained. The adaptive compression method means that: "If the character to be encoded exists in the current table, encode according to the table; otherwise, forward the ASCII code and update the table". Figure 3.13. illustrates adaptive coding.

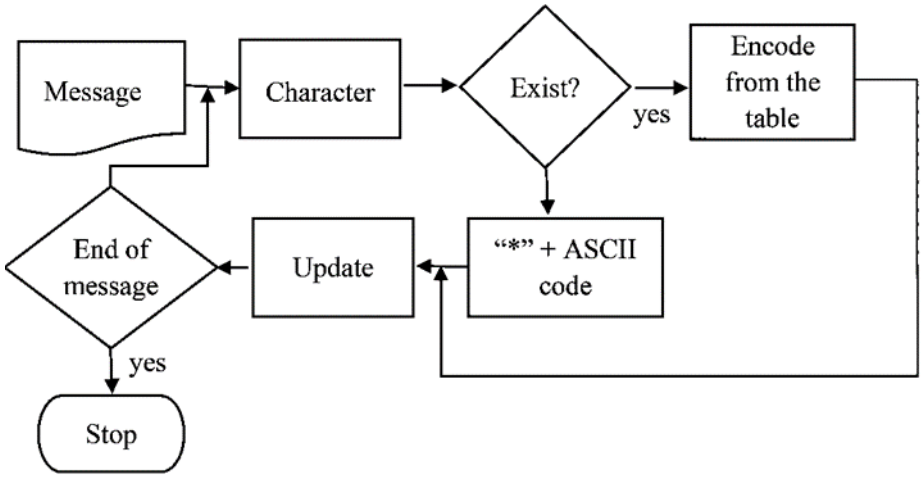


Figure 3.13. Algorithm of adaptive encoding

The A_ACHC algorithm works on the same principle. In the beginning, it starts with a table that contains only the "*" character. When a character is received from the message, if it is excluded from the code table, the code corresponding to the length of the * character is transmitted first, followed by the ASCII code. The character and its weight are added to the table, and the code lengths in the table are updated according to the ACHC algorithm. If the character is in the table, the code determined according to the corresponding length from the table is transmitted, and the weight of the character is increased by one. The code lengths in the table are updated by recalculating.

Compressing the M=”abbccda” message according to the A_ACHC algorithm is shown in Table 3.13.

Table 3.13. Steps of the A_ACHK Algorithm

s_i	W_i	e_k	e_i	d_j	l_i	t_m	N_j	U	Code
		0	1	0	0	1	1	0	
*	0	0	1	0	0	0	1	0	0

M=”a” output = **0 “a”**

(a)

s_i	W_i	e_k	e_i	d_j	l_i	t_m	N_j	U	Code
		0	1	0	0	1	1	0	
a	1	0	1	0	1	1	2	1	0
*	0	0	1	0	0	0	1	0	1

M=”ab” output = 0 “a”- **1 “b”**

(b)

s_i	W_i	e_k	e_i	d_j	l_i	t_m	N_j	U	Code
		0	1	0	0	1	1	0	
a	1	0	1	0	1	1	2	1	0
b	1	0,5	0,5	1	1	1	2	2	10
*	0	1	0	2	0	3	1	2	11

M=”abb” output = 0 “a” -1”b” -**10**

(c)

s_i	W_i	e_k	e_i	d_j	l_i	t_m	N_j	U	Code
		0	1	0	0	1	1	0	
b	2	0	1	0	1	1	2	1	0
a	1	0,7	0,3	1	1	1	2	2	10
*	0	1	0	2	0	3	1	2	11

M=”abbc” output = 0 “a”- 1”b” - 10 - **11 “c”**

(d)

s_i	W_i	e_k	e_i	d_j	l_i	t_m	N_j	U	Code
			0	1	0	0	1	1	0
b	2	0	1	0	1	1	2	1	0
a	1	0,5	0,5	1	1	1	2	2	10
c	1	0,75	0,25	2	1	1	2	3	110
*	0	0,75	0,25	3	0	0	1	3	111

M=”abbcc” output = 0 “a”- 1”b” -10 -11 “c” -

(e)

s_i	W_i	e_k	e_i	d_j	l_i	t_m	N_j	U	Code
		0	1	0	0	1	1	0	
b	2	0	1	0	1	1	2	1	0
c	2	0,4	0,6	1	1	1	2	2	10
a	1	0,8	0,2	2	1	1	2	3	110
*	0	0,8	0,2	3	0	0	1	3	111

M=”abbccd” output = 0 “a” - 1”b” -1 0 - 11 “c” – 110 - **111 “d”**

(f)

The lengths obtained at the end of the process are the same as those produced by the Vitter algorithm. The codes corresponding to the resulting lengths are canonical codes, as they are obtained by sequential binary addition. The weight of the * character is zero at all stages of the A_ACHC algorithm. However, different weight methods can also be applied. Here, w (*) =0 is taken for comparison with the FGK and Vitter algorithms. When calculating the * character, l=0 is taken. Although it has no effect on weight because it has a code length, the * character affects the average bit length per symbol, as in the FGK and Vitter algorithms. Therefore, in the standard Huffman and ACHC algorithms, the code lengths of the M message are L= {2,2,2,2}, while in the adaptive application they are L= {2,2,2,3,3}. The pseudocode of the A_ACHC algorithm is given below in Algorithm 3.2. where S is the alphabet used and W is the weight array. Because the content of S will change in each loop, the S array together with the W array is the input of the ACHC function described in Section 3.2.2.1. The code array is the canonical Huffman codes calculated according to the U array. Code calculation can also be performed in ACHC. In this case, the output of the ACHC function is the direct code array. The code example of the A_ACHC algorithm written in C is given in Appendix A. The code example of the Vitter algorithm is given in Appendix C.

```
A_CKHK()
…
Do while i< n                    // n is the number of symbols in the alphabet
{
        s=new character;                 // Get a character from the message
        If s is not in Table S
                Output ((Code (*) + ASCII(s)); // Send the ASCII code of s
                                                corresponding to the *
                                                character
                Add to the table;                // Add character s to table S
        If exist
                output (kod(s));     // Submit the corresponding binary s code
                                     from the code table
                W(s)=W(s)+1;     // Increase s's weight by one
        U=ACHC(W, U, S);       // Recalculate lengths on the basis of new
                               values
        Code=CalculateCode(U);         // Calculate canonical binary codes
                                       based on new lengths
```

Algorithm 3.2. A_ACHC Algorithm

Time and space complexity

If m is the number of preprocessed characters and the alphabet size is n, the time complexity required to encode a character is O(n-m). m is zero when all characters are processed. Therefore, since n will increase as 1+2+3+…+n, the total time complexity required to determine all codewords for A_ACHC will be n(n+1)/2≅ n^2. The time complexity required to encode a character in the FGK and Vitter algorithms is O(*l*). Considering all characters, *l* will increase from 1 to n in the worst case (the tallest tree). This makes O(n^2) the time required to determine all code words with the asymptotic approximation ((1+2+3+…+ (n-1))=n(n-1)/2≅n^2). This indicates that the A_ACHC algorithm is not slower in coding than the FGK and Vitter algorithms overall.

Calculating the time complexity of the A_ACHC algorithm is shown below:

Table.3.14. Time complexity of A_ACHC algorithm

Commands	**Cost**	**times**
Do while *i*< *n*	c_1	n+1
{		
s=new character;	c_2	n
If s is not in Table S // Table S is of length n	c_3	n * n
Output ((Code (*) + ASCII(s));	c_4	n
Add to the table // Added at the end of the table	c_5	n
If exist		
Output (kod(s));	c_6	n
W(s)=W(s)+1;	c_7	n
U=ACHC(W, U, S);	c_8	n * O(n)
}		
End		

$$T(n) = c_1 * (n+1) + c_2 * n + c_3 * n * n + c_4 * n + c_5 * n + c_6 * n + c_7 * n + c_8 * n * O(n)$$

$$T(n) = n * (c_1 + c_2 + c_4 + c_5 + c_6 + c_7) + n^2 * (c_3 + c_8) + c_1$$

$$A = c_3 + c_8 \qquad B = c_1 + c_2 + c_4 + c_5 + c_6 + c_7 \qquad C = c_1$$

$$T(n) = A * n^2 + B * n + C \quad => O(n^2)$$

The amount of memory required for A_ACHC is lower than that of the others. In the A_ACHC algorithm, three arrays must be used in the ACHC function: S for symbols, W for weights, and U for lengths, because of the requirement to be updated with each new symbol received. The number of bits required for n symbols is $n \log_2(n)$ for the S (alphabet) array. For the length array U, the character data type is sufficient, in which case the number of bits required is 8n for n symbols. The w weight array should hold the total number of characters (t) in the processed message. Therefore, the required number of bits for W is $n \log_2(t)$ bits for n symbols. The result is that the total number of bits required is $n \log_2(n) + n\ 8 + n \log_2(t)$ bits. This amount is lower than those of the FGK and Vitter algorithms. This is because the Vitter algorithm uses eight arrays, each of length 2n-1, while the A_ACHC algorithm uses three arrays of length n. The number of bits saved according to the commonly used Vitter algorithm can be calculated using the formula *T=1-A_ACHC(n,t)/Vitter(n,t)* (Equation 1.5.).

$$T = 1 - \left(\frac{n \log_2(n) + 8n + n \log_2(t)}{16n \log_2 n + 15n + 2n \log_2 t}\right) = 1 - \left(\frac{\log_2(n) + 8 + \log_2(t)}{16 \log_2 n + 15 + 2 \log_2 t}\right) \quad (4.3)$$

The demonstrating the T percentages according to different t values is shown in Figure 3.14. According to the Vitter algorithm, memory saving approaches 80% on average as n increases. This is an important advantage of the A_ACHC algorithm.

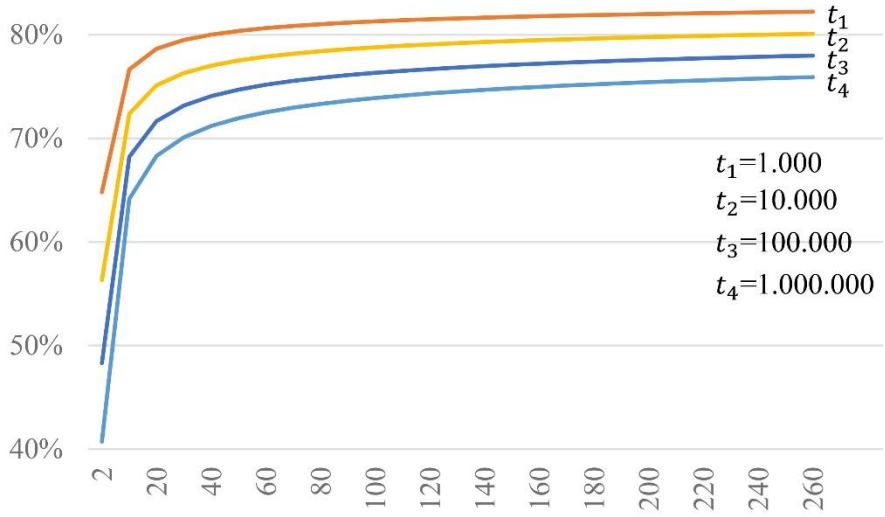


Figure 3.14. Amount of memory saved by the A_CKHK algorithm and the Vitter algorithm

*Weight issue of the * (NYT) character*

In adaptive coding, the * (NYT) character is used before the code of the new character to express the character encountered for the first time. Because it is excluded from the message to be transmitted, w (*) =0 is taken in the FGK and Vitter algorithms. Thus, the * character has no mathematical effect in the formula $A_v = \sum_{x=0}^{n} w_x l_x$ (Equation 3.6), which defines the average bit length per symbol. However, in the classical Huffman tree without the * character, while the code lengths for the message M=" abbccda" are L= {2,2,2,2}, in the adaptive algorithm, for example, in the Vitter and A_ACHC algorithms, the code lengths are L= {2,2,2,3,3}. This proves that, although it has zero weight, the presence of the * character alters the structure of the tree, thus the code lengths and the overall compressed file size. For this reason, tests were performed for w (*) =0, w (*) =1, and w (*) =x values with the A_ACHC algorithm. Tests have shown that a relatively shorter file size (better compression ratio) is attained using w (*) =x. The test results are shown in the fifth section (Table 4.5).

4. RESULTS AND DISCUSSIONS

4.1. ACHC Algorithm

ACHC, Polar codes, the standard Huffman algorithm (SHA), and Fyffe codes were tested in terms of average bit length per symbol. For each algorithm, the codeword lengths were first determined, and then the average bit lengths per symbol were calculated using Equation (3.6). All calculations are based on zero-order entropy. The Calgary Corpus was chosen for testing because it contains different types of files and is easily accessible by everyone. The test results are shown in Table 4.1. Since the codes produced by the standard Huffman algorithm (SHA) are optimal codes, comparisons were made between ACHC, Fyffe, and Polar codes that produce codes close to the optimum. The results show that ACHC is closest to the optimum SHA values for all files. The deviation percentages (error percentages) from SHA are shown in Table 4.2. Since the difference between SHA and ACHC is minimal, which of these algorithms to use is up to the user's choice (optimality, speed, and simplicity preference).

Table 4. 1. Bit lengths per symbol of the algorithms

Calgary	n	Size (Byte)	Entropy	SHA	ACHC	Fyffe	Polar
Bib	82	111,261	5.20079	5.23182	**5.23212**	5.30172	5.32792
Book1	83	768,771	4.52716	4.56183	**4.56257**	4.64724	4.64274
Book2	97	610,856	4.79265	4.82342	**4.82885**	4.94248	4.89313
Paper1	96	53,161	4.98321	5.01691	**5.01702**	5.10065	5.11203
Paper2	92	82,199	4.60159	4.63424	**4.63732**	4.73726	4.67905
Paper3	85	46,526	4.66536	4.68998	**4.69194**	4.78103	4.74075
Paper4	81	13,286	4.70051	4.73335	**4.73372**	4.83886	4.82659
Paper5	92	11,954	4.93699	4.97365	**4.97482**	5.02509	4.97984
Paper6	94	38,105	5.00980	5.04379	**5.04592**	5.11058	5.13186
News	99	377,109	5.18967	5.22702	**5.22975**	5.30306	5.23928
Geo	257	102,400	5.64649	5.66865	**5.66973**	5.73953	5.67935
Obj1	257	21,504	5.94863	5.97177	**5.97307**	6.02892	5.98432
Obj2	257	246,814	6.26043	6.29129	**6.29313**	6.37813	6.36367
Pic	160	513,216	1.21021	1.66094	**1.66157**	1.66861	1.66756
Progc	93	39,611	5.19930	5.23391	**5.23434**	5.31697	5.28264
Progl	88	71,646	4.77026	4.79954	**4.79960**	4.82840	4.85661
Progp	90	49,379	4.86901	4.89520	**4.89686**	4.96747	4.94317
Trans	100	93,695	5.53291	5.56861	**5.56935**	5.63264	5.58619

Some problems were encountered while compressing some files with Polar codes. For example, $s_n >> s_{n-1}$ case has occurred in "book2" and "news" files. Here, s is any symbol in the used alphabet and n is the number of symbols. In this case, the algorithm remains unsolved and enters the loop. R cannot reach zero. This issue was solved by taking $s_n = s_{n-1}$. Because in a complete binary tree structure, the last two elements must always be the same. Another problem was

encountered when compressing the “pic” file. Let the probability of any s character in the alphabet be 87%. Here, too, the length of the codeword associated with s is zero. Therefore, R=0 equality cannot be achieved while applying the algorithm. The solution to this problem is found by taking the length of the codeword of s as 1. In this case, contrary to what it should have been, R>0. In other words, the problem has been transformed into a situation to be solved by the Fyffe algorithm. The aforementioned problems and their solutions are not mentioned in the relevant link (Polar, 2010).

Table 4. 2. Error percentages of the ACHC, Fyffe, and Polar algorithms relative

Calgary	ACHC-SHA	Fyffe-SHA	Polar-SHA
Bib	**0.03%**	6.99%	9.61%
Book1	**0.07%**	8.54%	8.09%
Book2	**0.54%**	11.91%	6.97%
Paper1	**0.01%**	8.37%	9.51%
Paper2	**0.31%**	10.30%	4.48%
Paper3	**0.20%**	9.10%	5.08%
Paper4	**0.04%**	10.55%	9.32%
Paper5	**0.12%**	5.14%	0.62%
Paper6	**0.21%**	6.68%	8.81%
News	**0.27%**	7.60%	1.23%
Geo	**0.11%**	7.09%	1.07%
Obj1	**0.13%**	5.72%	1.26%
Obj2	**0.18%**	8.68%	7.24%
Pic	**0.06%**	0.77%	0.66%
Progc	**0.04%**	8.31%	4.87%
Progl	**0.01%**	2.89%	5.71%
Progp	**0.17%**	7.23%	4.80%
Trans	**0.07%**	6.40%	1.76%

4.2. ES_ACHC Algorithm

Among the results obtained with GA, only the “Bib” file in Calgary Corpus was used as an example. In the GA used, individuals were randomly generated. Roulette selection was used for selection, and crossover points were randomly selected using multi-point crossover depending on the alphabet size. Mutation points are also multiple and randomly selected from the locations where the length values change. The program was run 10 times for 11 mutation rate (MR) values for each crossover ratio, and the average of the acquired values is shown in bold font in Table 4.3. Because the related table is wide, it can be divided into two. The values shown in Table 4.3 show that the success of the algorithm is more dependent on the MR than on the crossover ratio. This indicates that mutation is more important in permutation coding sequence applications, such as Huffman code lengths.

The ESs test results are shown in Table 4.4. The average length value produced with the length array (first ancestor) closest to the best value obtained by the ACHC algorithm is shown in the ACHC column. With the evolution of this first ancestor, the length array that satisfies the best average length value (Huff_Av) is generated. The values of the parameters used in the algorithm were determined to give the best results after various trials. These parameters are: the mutation rate

(MR), number of mutation points (MP), and number of children and copies (λ). λ copies replicated from an ancestor mutate and form λ offspring. The values of the parameters, were determined like that: MR=0.95, MP=round(n/2), λ=5*n. MR=0.95 shows that 95% of the mutations occur at the transition points and 5% at the midpoints. The program was run 10 times for each file, and the number of loops in which it reached the best value in each run was noted. Finally, the average number of cycles is given in the last column. It is seen that the number of loops required to reach the best value has no relationship with parameters, such as the number of alphabets and file size. However, to reach the best value, it is adequate to take the number of loops to be used as fixed number, such as 100.

Table 4. 3. Average number of loops obtained with GA for the Bib file

		crossover ratio					
		0.0	0.1	0.2	0.3	0.4	0.5
mutation ratio	0	>10.000	>10.000	>10.000	>10.000	>10.000	>10.000
	0,1	3126	3458	3722	3670	2620	2793
	0,2	1314	2138	1828	2220	2023	2356
	0,3	2010	1229	611	1029	1755	1617
	0,4	819	1214	927	1568	695	1050
	0,5	579	719	666	546	219	966
	0,6	595	689	1009	392	535	1146
	0,7	501	514	212	284	360	571
	0,8	296	416	261	254	***177***	308
	0,9	336	***348***	***162***	***243***	348	292
	1	***101***	326	282	450	232	***275***

(a)

		crossover ratio				
		0.6	0.7	0.8	0.9	1.0
Mutation ratio	0	>10.000	>10.000	>10.000	>10.000	>10.000
	0,1	4186	4434	4854	4096	4893
	0,2	1629	2128	2838	2384	1516
	0,3	1434	1679	1355	1027	1403
	0,4	724	1190	1182	1508	1050
	0,5	474	754	1139	784	464
	0,6	779	487	656	693	695
	0,7	481	433	450	688	389
	0,8	600	199	518	***249***	554
	0,9	***140***	***90***	476	556	***245***
	1	205	230	***182***	269	391

(b)

Table 4. 4. ESs test results

Files	n	Size (Byte)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average Loop
bib	82	111.261	1	1	1	13	1	1	1	1	1	1	2,2
book1	83	768.771	30	34	5	44	38	9	48	42	47	63	36
news	99	377.109	3	3	2	2	5	1	2	3	12	5	3,8
paper1	96	53.161	2	2	1	1	2	1	1	1	2	1	1,4
paper2	92	82.199	9	14	9	8	7	11	11	6	7	6	8,8
progc	93	39.611	10	3	21	23	13	5	7	20	25	6	13,3
progp	90	49.379	59	5	1	34	32	1	64	69	12	1	27,8
trans	100	93.695	5	4	12	7	7	8	4	3	3	3	5,6
geo	257	102.400	10	5	4	6	5	5	5	4	5	5	5,4
obj1	257	21.504	5	5	7	6	6	6	6	7	6	5	5,9
obj2	257	246.814	9	9	8	6	9	8	7	9	8	11	8,4
pic	160	513.216	22	14	25	18	18	15	23	20	13	20	18,8

(a)

Dosya	n	Size (Bytr)	Huff_Av	ACHC (ancestor)
bib	82	111.261	5,2318	5,2320
book1	83	768.771	4,5618	4,5626
news	99	377.109	5,2270	5,2297
paper1	96	53.161	5,0169	5,0168
paper2	92	82.199	4,6342	4,6372
progc	93	39.611	5,2339	5,2341
progp	90	49.379	4,8952	4,8966
trans	100	93.695	5,5686	5,5692
geo	257	102.400	5,6687	5,6695
obj1	257	21.504	5,9718	5,9721
obj2	257	246.814	6,2913	6.2931
pic	160	513.216	1,6609	1,6615

(b)

4.3. A_ACHC Algorithm

Each file in Calgary Corpus was compressed with A_ACHC, FGK, and Vitter algorithms, and the results were compared. First, compression ratios were tested for different values of the * (NYT) character. These values are w (*) =0, w (*) =1, and w (*) =x. w (*) =0 is a zero weight, as in the FGK and Vitter algorithms. w (*) =1, has a weight of 1 in each case, and w (*) =x indicates the weight that changes according to the frequency of the * character. The results are shown in Table 4.5.

Table 4. 5. Comparison of compressed file sizes based on different values of the * character

	Original Size (Byte)	n	P(*)=x	P(*)=0	P(*)=1	0-x	1-x
paper4	13.286	80	***7.974***	7.982	7.981	8	7
bib	111.261	81	***72.890***	72.900	72.898	10	8
book1	768.771	82	***438.623***	438.635	438.632	12	9
paper3	46.526	84	***27.380***	27.389	27.388	9	8
progl	71.646	87	***43.115***	43.125	43.122	10	7
progp	49.379	89	***30.368***	30.378	30.377	10	9
paper2	82.199	91	***47.739***	47.749	47.748	10	9
paper5	11.954	92	***7.533***	7.542	7.541	9	8
progc	39.611	92	***26.046***	26.056	26.055	10	9
paper6	38.105	93	***24.158***	24.168	24.167	10	9
paper1	53.161	95	***33.504***	33.515	33.513	11	9
book2	610.856	96	***368.800***	368.811	368.808	11	8
news	377.109	98	***246.800***	246.812	246.810	12	10
trans	93.695	99	***65.386***	65.395	65.396	9	10
pic	513.216	159	***106.747***	106.773	106.775	26	28
geo	102.400	256	***72.867***	72.919	72.922	52	55
obj1	21.504	256	***16.372***	16.409	16.413	37	41
obj2	246.814	256	***194.552***	194.593	194.595	41	43

In Table 4.5., n is the alphabet size, "0-x" defines the difference between P (*) =0 and P (*) =x values, "1-x" shows the difference between P (*) =1 and P (*) =x values. Table 4.5 shows that as n increases, the choice of P (*) =x becomes more beneficial. This is an expected result. Because, as the frequency of the * character increases, it will have a shorter code than the less common characters. This causes the compressed file size to decrease. As a result, with n>1, choosing P (*) =x would be more helpful because n is larger for n-gram and word-based compression. The same advantage is seen in the compression of the measured values obtained from the sensors using A_ACHC or Vitter.

The A_CKHK algorithm was compared with the Vitter and FGK algorithms. Each file in Calgary Corpus is compressed using the A_CKHK algorithm, two Vitter algorithms, and two FGK algorithms, and the results are shown in Table 4.6.

The adaptive encoders used in the comparison were downloaded from https://github.com. The web addresses of the encoders are as follows:

Vitter1: https://github.com/santensuru/adaptive-huffman

Vitter2: https://github.com/gustavosobral/huffman

FGK1: https://github.com/beaverden/AdaptiveHuffman.git

FGK2: https://github.com/grtamayo/Algorithm-FGK

Compressed file sizes are calculated by multiplying the compression ratio by 100 (CRx100) shown in Equation (3.3). When the compression ratios in Table 4.6. are compared, the A_ACHC algorithm gives almost the same results as the other algorithms. In fact, A_ACHC yielded better results than the others for seven files did. Speed comparisons of the programs were not made because the speed also depends on the programing technique. As a result, Table 4.6. shows that the A_ACHC algorithm can be used instead of other algorithms in terms of the compression ratio.

Table 4.6. Comparison of the A_CKHK algorithm with the Vitter and FGK algorithms

	Original Size (Byte)	A_ACHC %	Vitter1 %	Vitter2 %	FGK1 %	FGK2 %
bib	111.261	65,5216	***65,5135***	65,7005	65,6097	65,5162
book1	768.771	57,0567	57,1270	57,0693	57,1144	***57,0413***
book2	610.856	60,3761	60,3740	60,3594	60,3471	***60,3222***
geo	102.400	***71,2100***	71,3076	71,9648	71,3398	71,2334
news	377.109	65,4485	65,4161	65,4463	65,3917	***65,3885***
obj1	21.504	***76,3067***	76,6276	80,0781	76,4276	76,3346
obj2	246.814	78,8420	78,9052	79,0911	79,6746	***78,8217***
paper1	53.161	63,0443	63,0556	63,4939	63,0142	***62,9917***
paper2	82.199	***58,0895***	58,1126	58,4289	58,1406	58,1065
paper3	46.526	***58,8682***	59,0745	59,4356	59,1325	58,9047
paper4	13.286	60,0783	60,0858	61,8997	60,0933	***60,0708***
paper5	11.954	***63,0919***	63,3512	65,6684	63,3679	63,3010
paper6	38.105	***63,4247***	63,5481	64,1438	63,4667	63,4405
pic	513.216	20,8047	20,8076	20,9041	20,8224	***20,8035***
progc	39.611	***65,7797***	65,8226	66,4386	65,8352	65,8024
progl	71.646	60,1918	60,3258	60,5323	60,2518	***60,1862***
progp	49.379	61,5201	61,5322	61,9838	61,5282	***61,4836***
trans	93.695	69,7956	69,7988	70,0688	69,8522	***69,7913***

5. CONCLUSION

Three algorithms are discussed in this thesis. The first is the ACHC algorithm, which calculates the canonical Huffman code lengths. The second algorithm is the ES_ACHC algorithm, which obtains the optimum Canonical Huffman code lengths with ESs and optimizes the ACHC algorithm. As Finally, A_ACHC, which is the adaptive version of the ACHC algorithm.

ACHC has been devised as a fast algorithm for generating canonical prefix-free code. With this algorithm, the lengths of "prefix-free" canonical code words are determined by algebraic calculation according to their weights. The code words are then obtained in canonical form using these lengths. While the time complexity of SHA is O (*nlogn*), the time complexity of ACHC is Θ (n). The space complexity of ACHC is Θ (*n*) word memory. Since the amount of memory consumed for the standard Huffman algorithm is 5n words, the memory savings achieved with the ACHC algorithm is 80%. However, the results are usually not optimal, but are so close to the optimum. The results of the tests showed that the difference between SHA and ACHC is quite small compared with their counterparts.

The ACHC algorithm is very simple and easy to implement and has shorter lines of code than Huffman coding, as shown in the Appendix B. Considering its speed, simplicity, and the amount of memory it uses, ACHC can be preferred in various applications, such as HTML pages and online games. ACHC may also be preferable for larger alphabets, where speed may be more important than the compression ratio. Here, the difference between SHA and ACHC is negligible in terms of the compressed stream size.

An algorithm that produces optimal Canonical Huffman codes different from the traditional method has not been found in the literature. However, there are algorithms (Fyffe, Polar) that produce near-optimal codes. ES_ACHC is an algorithm that obtains the canonical Huffman code lengths through evolutionary algorithms, unlike the traditional method. Unlike previous studies on data compression, the evolutionary algorithm used is the fixed parameter evolution strategies algorithm, and the mutation method is the target-directed mutation. With this method, the code lengths (bit numbers) that form the basis for the generation of Canonical Huffman codes are obtained at so low loop time. Canonical codes are derived from these lengths. After the canonical codes are obtained, the text to be compressed is compressed using these codes. Since the ACHC algorithm, which produces the sequence with the closest length to the best value, is used as the ancestor, this proposed and examined method also optimize the ACHC algorithm. As a feature of heuristic methods, because multiple individuals are used in the search space, the speed of reaching the result is slow ($O(n^2)$) and the amount of memory used ($O(n^2)$) is high compared to traditional methods. However, considering the size of the file to be compressed and current computer technology, the speed of the algorithm will not be a problem.

For permutation coding applications, this study demonstrates that evolution strategies with constant parameters may be a more suitable choice for researchers besides the genetic algorithm. The method used in the research of Zaki and Sayed mentioned in the second chapter deteriorates the prefix-free feature in the tree structures because of crossover and mutation (Zaki and Sayed, 2009). To repair these deteriorated individuals, this thesis proposed the "Fyffe tuning" technique. In addition, the ES algorithm used in this study, unlike the known models, is an ES model whose strategy parameter does not change. This new model is named "ESs with Constant Strategy Parameter, (1 + λ)–CSP-ES".

The ES_ACHC algorithm, which is character-based (monogram), is the first step for syllable-based (n-gram) and word-based alphabets and can be easily adapted to syllable-based alphabets. For this reason, the next study will focus on obtaining hybrid alphabets and codes, which are syllables, words, or mixtures of these, with ES_ACHC.

In this thesis, the adaptive (dynamic) application of the ACHC algorithm (A_ ACHC), which calculates the canonical Huffman code lengths algebraically, is also discussed. Using the A_ACHC algorithm, it has been shown that while providing the same compression ratios, 80% less memory usage can be achieved compared with the Vitter and FGK algorithms. Furthermore, the A_ACHC algorithm has shorter lines of code than Vitter and similar algorithms, as shown in the Appendices. Because of memory savings and shorter code implementation, the proposed algorithm is considered to be more suitable for text and image compression, as well as for data compression in sensors and wearable devices used in wireless sensor networks.

In addition, for the character *, which represents previously unseen characters, the effect of zero weight (p (*) =0), fixed 1 weight (p (*) =1), and varying weight (p (*) =x) options on compression was investigated. The case of P (*) =x has been shown to provide better compression, especially as n increases. Therefore, because n is much larger in n-gram and word-based applications, as well as in measurement-based applications, it will give better results to evaluate the weight of the * character according to its frequency of occurrence.

REFERENCES

Abuzanouneh, K. I., 2017. Develop and Design Hybrid Genetic Algorithms with Multiple Objectives in Data Compression. IJCSNS International Journal of Computer Science and Network Security, 17(10):32-39.

Auger, A., 2005. Convergence results for the (1,λ)-SA-ES using the theory of φ-irreducible Markov chains. Theoretical Computer Science, 334:35-69.

Auger, A., Hansen, A. N., 2005. A Restart CMA Evolution Strategy with Increasing Population Size,. IEEE Congress on Evolutionary.

Barbay, J., 2016. Optimal Prefix Free Codes with Partial Sorting. 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), 29: 1–13.

Boisclair, C., Wagner, M., 2008. Better Huffman Coding via Genetic Algorithm. The Proceedings of the 2008 International Conference on Genetic and Evolutionary Methods, GEM 2008, 30-34.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. 2009. Introduction to Algorithms, 3th edition. Massachussetts: The MIT Press. 1292p

Dube, D., Beaudoin, V., 2008. Fast Construction of Disposable Prefix-Free Codes. Comptes-rendus du International Colloquium on Signal Processing and its Applicationa, 49-54.

Gajjala et al, 2020. Huffman Coding Based Encoding Techniques for Fast Distributed Deep Learning. Proceedings of the 1st Workshop on Distributed Machine Learning. Barcelona, Spain. doi: 10.1145/3426745.3431334.

Gallager, R. G., 1978. Variations on a theme by Huffman. IEEE Transactions on Inforn Theory, 24(6):668-674.

Geldreich, R., 1999. lzham-Fyffe Codes.wiki. 10 16, 2022: https://code.google.com/archive/p/lzham/wikis/FyffeCodes.wiki.

Günter, R., 2000. 9: Evolution Strategies. Evolutionary Computation 1: Basic Algorithms and Operations. 81-88. in Bristol: Institute of Physics Publishing.

Hidayat, T., Zakaria, M. H., Pee, A. N. C., 2023. Increasing the Huffman generation code algorithm to equalize compression ratio and time in lossless 16-bit data archiving. Multimedia Tools and Applications, 82(16):24031-24068, doi: 10.1007/s11042-022-14130-1.

Hosseini, M., 2012. A Survey of Data Compression Algorithms and their Applications. doi:10.13140/2.1.4360.9924.

Hromada, V., Grošek, O., 2022. A Note on the Maximum Size of a Prefix Code. IEEE Access, 10: 125955-125962, doi: 10.1109/ACCESS.2022.3222354.

Huffman, D. A., 1952. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE, 40(9):1098 - 1101. doi:10.1109/JRPROC.1952.273898.

Kalaivani, S., Tharini, C., 2020. Analysis and Implementation of Novel Rice Golomb Coding Algorithm Wireless Sensor Networks. Computer Communications, 150:463-471. doi: https://doi.org/10.1016/j.comcom.2019.11.046.

Kattan, A., Poli, R., 2008a. Evolutionary lossless compression with GP-ZIP. 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence, 2468-2472.

_____, 2008b. Evolutionary Lossless Compression with GP-ZIP*. In GECCO'08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, 2008. Atlanta.

_____, 2009. Genetic-Programming Based Prediction of Data Compression Saving. 9th International Conference on Artificial Evolution (Evolution Artificielle), 182-193.

_____, 2010. Evolutionary Synthesis of Lossless Compression Algorithms with GP-ZIP3. IEEE Congress on Evolutionary Computation, 1-8.

_____, 2011. Evolutionary synthesis of lossless compression algorithms with GP-zip2. Genetic Programming and Evolvable Machines, 12(4):335–364. doi:10.1007/s10710-011-9133-6.

Klein, S. T., Saadia, S., Shapira, D., 2021. Forward Looking Huffman Coding. Theory of Computing Systems, 65(3):593-612. doi:10.1007/s00224-020-09992-7.

Klein, S. T., Shapira, D., Fruchtman, A., 2019. Bidirectional Adaptive Compression. Proceedings of the Prague Stringology Conference 2019, 92-101.

Klus, L., Lohan, E. S., Granell, C., & Nurmi, J. 2020. Lossy compression methods for performance-restricted wearable devices. WiP Proceedings of the International Conference on Localization and GNSS (ICL-GNSS 2020), Tempere, Finland.

Knuth, D. E., 1985. Dynamic Huffman coding. Journal of Algorithms, 6:163-180.

Lánský, J., Kuthan, T., 2007. Genetic algorithms in syllable based text compression. Proceedings of the Dateso 2007 Annual International Workshop on DAtabases, TExts, Specifications and Objects, 235:21-34.

Leeuwen, J., 1976. On the Construction of Huffman Trees. ICALP, 382-410.

Lu, W.-W., & Gough, M. P., 1993). A Fast Adaptive Huffman Coding Algorithm. IEEE Transactions on Communications, 41(4).

Marjai, Péter, Péter Lehotay-Kéry, and Attila Kiss. 2022. A Novel Dictionary-Based Method to Compress Log Files with Different Message Frequency Distributions. Applied Sciences. 12(4): 2044. https://doi.org/10.3390/app12042044.

Moffat, A., 2020. Huffman Coding. ACM Computing Surveys, 52(4):1-35. doi:10.1145/3342555.

Moffat, A. T., 1997. On the Implementation of Minimum Redundancy Prefix Codes. IEEE Transactions on Communications, 45(10):1200-1207.

Moffat, A., Katajainen, J., 1995. In-place calculation of minimum-redundancy codes. Proceedings of the Workshop on Algorithms and Data Structures, 393–402.

Moffat, A., Petri, M., 2020. Large-Alphabet Semi-Static Entropy Coding Via Asymmetric Numeral Systems. ACM Transactions on Information Systems, 38(4):1-33, doi: 10.1145/3397175.

Moffat, A., Turpin, A., 1998. Efficient construction of minimum-redundancy codes for large alphabets. IEEE Transactions on Information Theory, 44(4):1650-1657.

Navarro, G. A., 2013. Compressing Huffman models on large alphabets. Proc. 23rd Data Compression Conference (DCC), 381-390.

Oral, M., Aşşık, M. M., 2019. An Algorithm that Calculates the Lengths of Codewords Algebraically for Canonical Huffman-like Encoding. Cukurova University Journal of the Faculty of Engineering and Architecture, 34(4):9-20.

Oroumchian, F., Darrudi, E., Taghiyareh, F., Angoshtari, N., 2004. Experiments with persian text compression for web. Proceeding of the 13th international World Wide Web conference on alternate track papers & posters, WWW Alt. '04, 2004. 478-479).

Platos, J., Krömer, P., 2011. Evolving alphabet using genetic algorithms. Proceedings of the 2011 3rd World Congress on Nature and Biologically Inspired Computing, 575 –581, doi:10.1109/NaBIC.2011.6089652.

_____, 2011. Optimizing alphabet using genetic algorithms. International Conference on Intelligent Systems Design and Applications, ISDA, 189:498-503, doi:10.1109/ISDA.2011.6121705.

_____, 2012. Searching for Optimal Alphabet for Data. IEEE International Conference on Systems, Man, and Cybernetics, 468-473.

Polar, A., 2010. Non-Huffman Binary tree. In 10-16-2022, http://www.ezcodesample.com/prefixer/prefixer_article.html.

Ranjan,R., Kumar, P., 2023. An Improved Image Compression Algorithm Using 2D DWT and PCA with Canonical Huffman Encoding. Entropy, 225(10):1382, https://doi.org/10.3390/e2510138.

Rioul, O, 2021. This is IT: A Primer on Shannon's Entropy and Information in Information Theory: Poincaré Seminar 2018. (B. Duplantier, & V. Rivasseau, Dü) Springer International Publishing, p.49-86, 209 p, Doi: 10.1007/978-3-030-81480-9_2.

Sayood, K., 2018. Introduction to Data, Fifth Edition, Morgan Kaufmann publications. Cambridge, 765p.

_____., 2018. Introduction to Data, Fifth Edition, Morgan Kaufmann publications. Cambridge, 11-17.

Shahbahrami A., E. A., 2011. Evaluation of Huffman and Arithmetic Algorithms for Multimedia Compression Standards. International Journal of Computer Science, Engineering and Applications (IJCSEA), 1(4).

Shannon, C. E., 1948. A mathematical theory of communication. The Bell System Technical Journal, 27(4):623-656.

Solomon, D., Motta, G., 2010. Handbook of Data Compression, 5th edition. Springer Publishing, London, 1359p.

Tharini, C., Vanaja Ranjan, P., 2009. Design of Modified Adaptive Huffman Data Compression. Journal of Computer Science, 5(6):466-470. doi:https://doi.org/10.3844/jcssp.2009.466.470

The Calgary Corpus., 2022. (M. Powell, Producer & University of Canterbury) ,in 2022, https://corpus.canterbury.ac.nz/descriptions/index.html#calgary

Usama, M., Malluhi, Q. M., Zakaria, N., Razzak, I., & Iqbal, W., 2021). An efficient secure data compression technique based on chaos and adaptive Huffman coding. Peer-to-peer Network Applications, 14:2651-2664. doi: https://doi,org/10.1007/s12083-020-00981-8.

Üçoluk, G., Toroslu, H., 1997. Genetic algorithm approach for verification of the syllable based text compression technique. Computer Journal of Information Science,23(5): 365-372.

Vitter, J. S., 1987. Design and Analysis of Dynamic Huffman Codes. Journal of the ACM, 34(4):825–845. doi: https://doi.org/10.1145/31846.42227

Zaki, M., Sayed, M., 2009. The use of genetic programming for adaptive text compression. Int. J. Information and Coding Theory, 1(1):88–108.

CURRICULUM VITAE

Personal Information

Surname, Name: AŞŞIK, Muhammed Mustafa

Education

Degree	Institution	Graduation Year
Postgraduate	Cukurova University	1995
Undergraduate	Erciyes University	1986
High School	Kdz. Eregli Lisesi	1979

Experience

Year	Work Place	Title
2010-Present	Cukurova University	Telephone Exchange Branch Manager
2000-2010	Turk Telekom	Payphone Directorate, Manager
1993-2000	Turk Telekom	Payphone Directorate, Assistant Technical Manager
1987- 1993	P.T.T	Telephone Exchange Engineer

Publications

1. Oral, M., Aşşık, M. M., 2019. An Algorithm that Calculates the Lengths of Codewords Algebraically for Canonical Huffman-like Encoding. Cukurova University Journal of the Faculty of Engineering and Architecture, 34(4):9-20.
2. Aşşık M. M and Oral M. , " Determination of canonical Huffman codeword lengths by evolution strategies ", Journal of the Faculty of Engineering and Architecture of Gazi University, vol. 38, no. 2, pp. 771-780, Oct. 2022, doi:10.17341/gazimmfd.882745

APPENDICES

APPENDIX A. A PROGRAM EXAMPLE OF A_ACHC ALGORITHM

```
…
Int main (int argc, char *argv[])
…
Parameter definitions, opening files for reading and writing and other operations
…
//Reading character and coding
for (i=1;i<Dboyut;i++)    // Dboyut =Size of the file
  {
    Krktr=Metin[i];   // Krktr=character, Metin=File to be encoded
    for (indc=0 ; indc<LAlf ; indc++)  // LAlf= size of the alphabet
    {
      if (Krktr==Alf[indc])    //Alf=Alphabet, Krktr is searched in Alf
        {  Ynt=1;   break; }  // if exist in Alf
      else
        {  Ynt=0; }  //if  Krktr is not exist in ALF
    }
    if (Ynt==0)
    {   for(t=0;t<strlen(Code[NYT]) ; t++)
          {
            Comp[Uz_Comp]=Code[NYT][t];       //NYT is added
            Uz_Comp++;
          }
          dec2bin(Krktr,1,8,Bin);            // Krktr is converted to binary
          for(t=0;t<8;t++)
          {
            Comp[Uz_Comp]=Bin[t];       //Comp=compressed file
            Uz_Comp++;
          }
         LAlf=Tablo_Guncelle(Alf,Freq,LAlf,Krktr);   // updating the table
    }
    else
    {
          for(t=0;t<strlen(Code[indc]);t++)
          {
            Comp[Uz_Comp]=Code[indc][t];    // added the code corresponding // to the character
            Uz_Comp++;
```

```
            }
            Freq[indc]=Freq[indc]+1;
      }
      Top_Freq=Top_Freq+1;   // Total frequence value
      for (z=0;z<LAlf;z++)
      {   P[z]=(double)Freq[z]/(double)Top_Freq;   }  // calculating probabilities
      Encoding(LAlf,P,U);  // ACHC function, Output is the length array (U)
      CodeU(LAlf,U,Bin,Code);  // calculate canonical codes from lengths
   }
```

ACHC Function

```
void Encoding (int boy, double *p, int *u)
{ // boy = size of the alphabet, p = probabilities, u = lengths
 //Parameters
 int s=-1;        // Specifies the reference node where the operation is performed
 int l=0;         // distance from the reference node being transacted, previous value
 int lx=0;        // distance from the reference node being processed, current value
 int n=1;         // number of leaves of the processed branch, previous value
int nx=1;         // number of the leaf being processed, current value
 int n1=0;     // maximum number of branches in the processed branch, previous value
int n1x=0;     // maximum number of branches in the processed branch, current value
double eksT=0;      // Cumulative sum of possibilities double
eks=1;     // Total probability value at the node where l will be measured
double Peks;
 int i;
// Begin
   for (i=0;i<boy;i++)
   {
      if (n==1)
      {   eks = 1 - eksT; }
      lx=round (log2 (eks / ( *(p+1 ))));
      Peks=*(p+i) / eks;
      if (i! = (boy-1) && Peks > 0.7)
      { lx = 1; }
      n1x = pow(2,(lx-1));
      if (n == 1)
      {  nx = n1x ;  s=s+1 ;  }
```

```
    else
    {
       if ( l < lx )
       {  nx = n1x - (n1-n + 1) * pow(2,(lx-l)); }
       else { nx = n-1 ; }
    }
    *(u+i) =  s + lx;
    L = lx;     n = nx;    n1 = n1x;    eksT = eksT + *(p+i);
}
}
```

APPENDIX B. A PROGRAM EXAMPLE OF HUFFMAN ALGORITHM

This program example is taken from "https://rosettacode.org/wiki /Huffman_coding#C". Encodes the message "this is an example for Huffman encoding" with the standard Huffman algorithm.

Using a simple heap-based priority queue. Heap is an array, while ndoe tree is done by binary links.

```
#include <stdio.h>
#include <string.h>
typedef struct node_t {
        struct node_t *left, *right;
        int freq;
        char c;
} *node;
struct node_t pool[256] = {{0}};
node qqq[255], *q = qqq - 1;
int n_nodes = 0, qend = 1;
char *code[128] = {0}, buf[1024];
node new_node(int freq, char c, node a, node b)
{
        node n = pool + n_nodes++;
        if (freq) n->c = c, n->freq = freq;
        else {
          n->left = a, n->right = b;
          n->freq = a->freq + b->freq;
        }
        return n;
}

/* priority queue */
void qinsert(node n)
{
        int j, i = qend++;
        while ((j = i / 2)) {
          if (q[j]->freq <= n->freq) break;
          q[i] = q[j], i = j;
        }
        q[i] = n;
```

```
}
node qremove()
{
        int i, l;
        node n = q[i = 1];
        if (qend < 2) return 0;
        qend--;
        while ((l = i * 2) < qend) {
          if (l + 1 < qend && q[l + 1]->freq < q[l]->freq) l++;
          q[i] = q[l], i = l;
        }
        q[i] = q[qend];
        return n;
}
/* walk the tree and put 0s and 1s */
void build_code(node n, char *s, int len)
{
        static char *out = buf;
        if (n->c) {
          s[len] = 0;
          strcpy(out, s);
          code[n->c] = out;
          out += len + 1;
          return;
        }
        s[len] = '0'; build_code(n->left,  s, len + 1);
        s[len] = '1'; build_code(n->right, s, len + 1);
}
void init(const char *s)
{
        int i, freq[128] = {0};
        char c[16];
        while (*s) freq[(int)*s++]++;
        for (i = 0; i < 128; i++)
          if (freq[i]) qinsert(new_node(freq[i], i, 0, 0));
        while (qend > 2)
          qinsert(new_node(0, 0, qremove(), qremove()));
```

```
        build_code(q[1], c, 0);
}
void encode(const char *s, char *out)
{
        while (*s) {
          strcpy(out, code[*s]);
          out += strlen(code[*s++]);
        }
}
void decode(const char *s, node t)
{
        node n = t;
        while (*s) {
          if (*s++ == '0') n = n->left;
          else n = n->right;
          if (n->c) putchar(n->c), n = t;
        }
        putchar('\n');
        if (t != n) printf("garbage input\n");
}
int main(void)
{
        int i;
        const char *str = "this is an example for huffman encoding";
       char buf[1024];
        init(str);
        for (i = 0; i < 128; i++)
          if (code[i]) printf("'%c': %s\n", i, code[i]);
        encode(str, buf);
        printf("encoded: %s\n", buf);
        printf("decoded: ");
        decode(buf, q[1]);
        return 0;
}
```

Output:

```
' ': 000
'a': 1000
```

'c': 01101

'd': 01100

'e': 0101

'f': 0010

'g': 010000

'h': 1101

'i': 0011

'l': 010001

'm': 1111

'n': 101

'o': 1110

'p': 10011

'r': 10010

's': 1100

't': 01111

'u': 01110

'x': 01001

encoded:

011111101001111000000011110000010001010000101010011000111110011010001010100000101110100100001101011100010001011111000101000010110101101111001100001110101000 0

decoded: this is an example for huffman encoding

APPENDIX C. A PROGRAM EXAMPLE OF VITTER ALGORITHM

This program example was written by Jeffrey Scott Vitter. The program is in Pascal. Cited publication: Algorithm 673: Dynamic Huffman Coding, ACM Transactions on Mathematical Software, Vol. 15, No. 2, June 1989, Pages 158-167.

```
Initialize;
while there are more letters to encode do begin
Let aj be the next letter to encode;
EncodeAndTransmit ( j );
Update( j )
end;
procedure Initialize;
var i : integer;
begin
M := 0; E := 0; R := -1; Z := 2 X n - 1;
For i:=l to n do begin
M:=M+l;R:=R+l;
İf 2xR=M then begin E:=E+l; R:=O end;
alpha[i] := i; rep[i] := i
end;
(Initialize node n as the O-node)
block[n] := 1; prevBlock[l] := 1; nextBlock[l] := 1; weight[l] := 0;
first[l] := n; last[l] := n; parity[l] := O; parent[l] := 0;
(Initialize available block list)
availBlock : = 2;
for i := availBlock to Z - 1 do
nextBlock[i] := i + 1;
nextBlock[Z] : = 0
end;
procedure EncodeAndTransmit ( j: integer);
var i, ii, q, t, root: integer;
begin
q := rep[j]; i := 0;
if q < M then begin (Encode letter of zero weight)
q := q - 1;
i fq<2XR then t:= E+l else begin q:=q-R; t:=E end;
```

```
for ii := 1 to t do begin
i := i + 1; stack[i] := q mod 2;
qz; q div 2
q:=M;
end;
if M = n then root := n else root := 2;
while q # root do begin {Traverse up the tree)
i := i + 1; stack[i] := (first[block[q]] - q + parity[block[q]]) mod 2;
q :=parent[block[q]] - (first[block[q]] - q + 1 -parity[block[q]]) div 2
end;
for ii := i downto 1 do Transmit (stuck[ii])
end;
procedure Update (k : integer);
var q, leafToIncrement, bq, b, oldparent, oldParity, nbq, par, bpar : integer;
    slide: boolean;
begin
{Set q to the node whose weight should increase)
FindNode;
while q > 0 do
(At this point, q is the first node in its block. Increment q's weight by 1 and slide q
if necessary over the next block to maintain the invariant. Then set q to the node
one level higher that needs incrementing next]
SlideAndIncrement ;
(Finish up some special cases involving the O-node]
if leafToIncrement # 0 then
begin q : = leafToIncrement ; SlideAndIncrement end
end;
procedure FindNode;
begin
q := rep[k]; 1eafToIncrement := 0;
if q = M then begin {A zero weight becomes positive)
InterchangeLeaves( q, M);
İf R=O then begin R:=M div.2; if  R>Othen E:=E-1 end;
M:= M - 1; R := R - 1; q := M + 1; bq := block[q];
if M > 0 then begin
(Split the O-node into an internal node with two children. The new O-node is
node M; the old O-node is node M + 1; the new parent of nodes M and M + 1 is
```

```
node M+ n)
block[M] := bq; last [bq] := M; oldParent :=parent[ bq];
 parent[bq] := M + n; parity [bq] := 1;
(Create a new internal block of zero weight for node M + n)
b : = availBlock; availBlock : = nextBlock[availBlock];
prevBlock[b] := bq; nextBlock[b] := nextBlock[bq];
prevBlock[nextBlock[bq]] := b; nextBlock[bq] := b;
parent[b] := oldParent; parity[b] := 0; rtChild [b] := q;
block [M + n] := b; weight         [b] := 0;
first [b] := M + n; last [b] := M + n;
leafToIncrement := q; q : = M + n
      end
end
else begin (Interchange q with the first node in q's block)
InterchangeLeaves ( q, first[block[ q]]);
q := first[block [ q]];
if (q=M+l) and (M>O) then
begin leafToIncrement := q; q := parent[block[ q]] end
end
end;
procedure SlideAndIncrement ;
begin ( q is currently the first node in its block)
bq := block [q]; nbq := nextBlock [bq];
par : = parent [bq]; oldParent : = par; oldParity : = parity [bq];
if ((q <= n) and (first [nbq ] > n) and (weigh [nbq] = weight [bq]))
or ((q > n) and (first [nbq] <= n) and (weight [nbq] = weight [bq] + 1)) then
begin (Slide q over the next block)
slide : = true;
oldParent := parent [nbq]; oldParity := parity [nbq];
(Adjust child pointers for next higher level in tree)
if par > 0 then begin
bpar : = block[par];
if rtChild [bpar] = q then rtChild [bpar] := last [nbq]
else if rtChild [bpar] = first [nbq] then rtChild [bpar] := q
else rtChild [bpar] : = rtChild [bpar] + 1;
if par # Z then
if block[par + l] # bpar then
```

```
if rtChild[block[par + l]] = first[nbq] then
rtChild[block[par + l]] := q
else if block[rtChild [block[par + l]]] = nbq then
rtChild[block[par + l]] : = rtChild [block[par + l]] + 1
end;
(Adjust parent pointers for block nbq )
 Parent [nbq] := parent [nbq] - 1 + parity [nbq]; parity [nbq] := 1 – parity [nbq];
nbq := nextBlock[nbq];
end;
else slide := false;
if (((q <= n) and (first [nbq] <= n)) or ((q > n) and (first [nbq] > n)))
and (weight [nbq] = weight [bq] + 1) then
begin (Merge q into the block of weight one higher)
block [q] := nbq; last [nbq] := q;
if last [bq] = q then begin (q's old block disappears)
nextBlock[preuBlock [bq]] := nextBlock [bq];
preuBlock[nextBlock [bq]] := preuBlock [bq];
nextBlock [bq] : = auailBlock; auailBlock : = bq
end
else begin
if q > n then rtChild [bq] := FindChild (q - 1, 1);
if parity [bq] = 0 then parent [bq] := parent [bq] - 1;
parity [bq] := 1 – parity [bq];
first [bq] := q - 1
end
end
else if last [bq ] = q then begin
if slide then begin (q's block is slid forward in the block list)
prevBlock [nextBlock [bq]] := prevBlock [bq];
nextBlock [prevBlock [bq]] := nextBlock [bq];
prevBlock [bq] := prevBlock [nbq]; nextBlock [bq] := nbq;
prevBlock [nbq] := bq; nextBlock [prevBlock [bq]  := bq;
parent [bq] := oldParent; parity [bq] := oldParity
end;
weight [bq] := weight [bq] + 1
end
else begin ]A new block is created for q]
```

```
b := availBlock; availBlock := nextBlock [availBlock];
block[q] := b; first[b] := q; last[b] := q;
if q > n then begin
rtChild [b] := rtChild [bq];
rtChild [bq] := FindChild (q - 1, 1);
if rtChild [b] = q - 1 then parent [bq] := q
else if parity [bq] = 0 then parent [bq] := parent [bq] - 1
end
else if parity[ bq] = 0 then parent [bq] := parent [bq] - 1;
first [bq] := q - 1; parity[ bq] := 1 – parity [bq];
{Insert q's block in its proper place in the block list)
prevBlock [b] := prevBlock [nbq]; nextBlock[ b] := nbq;
 prevBlock[nbq] := b; nextBlock [prevBlockrz [b]] := b;
weight [b] := weight [bq] + 1;
parent [b] : = oldParent ; parity [b] : = oldParity
end;
{Move q one level higher in the tree)
if q <= n then q := oldParent else q := par
end;
```