University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Marek Biskup

Error Resilience in Compressed Data — Selected Topics

PhD dissertation

Supervisor

dr hab. Wojciech Plandowski, prof. UW

Institute of Informatics
University of Warsaw

September 2008

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

September 9, 2008 .....
date *Marek Biskup*

Supervisor's declaration:
the dissertation is ready to be reviewed

September 9, 2008 .....
date *dr hab. Wojciech Plandowski, prof. UW*

Abstract

In compressed data a single bit error propagates because of the corruption of the decoder's state. This work is a study of error resilience in compressed data and, in particular, of the recovery of as much data as possible after a bit error. It is focused on Huffman codes.

In a message encoded with a Huffman code a bit error causes the decoder to lose synchronization with the coder. The error propagates because the codewords seen by the decoder are misaligned. In case of most Huffman codes the decoder eventually resynchronizes. Nevertheless, there is no a priori upper bound on the number of incorrectly decoded symbols.

The work introduces two novel methods for limiting error propagation in Huffman codes to not more than L bits, L being a parameter. In one method it is assumed that the decoder knows its position in the encoded message, in the other, the position of the decoder may be unknown. The methods are based on synchronization of a decoder that starts at an arbitrary bit of the encoded data. They utilize the inherent tendency of Huffman codes to synchronize spontaneously and do not introduce any redundancy if such a synchronization takes place. Another new method for limiting error propagation, presented in this dissertation, can be used in a wide class of codes. The methods are applied to parallel decoding of Huffman data and are tested on Jpeg compression. Additionally, an algorithm for finding correct codeword's alignment by a decoder that starts in the middle of a message encoded with normal Huffman coding is presented.

Statistical synchronization of Huffman codes is related to synchronizing strings — strings that always resynchronize the decoder. It is shown that finding a synchronizing string for a code is equivalent to a finding a synchronizing string for some finite automaton. Černý conjecture for this class of automata is discussed and an upper bound on the length of the shortest synchronizing string is presented. It is supported by an efficient algorithm that checks if a code has a synchronizing string and, if so, constructs a synchronizing string achieving the bound. Two classes of codes with a long shortest synchronizing string are shown with an exact length of the shortest synchronizing string. Finally, two efficient algorithms for finding all the synchronizing codewords — synchronizing strings that are codewords — of a Huffman code are presented.

Keywords: Huffman code, synchronization, synchronization delay, guaranteed synchronization, synchronizing string, synchronizing codeword, resynchronization marker, error resilience, strong synchronization.

ACM Classification: E.4, F.1.1.

Streszczenie

Błąd pojedynczego bitu w skompresowanych danych propaguje się, ponieważ zmienia on stan dekodera. W pracy analizowana jest odporność na błędy w skompresowanych danych, w szczególności problem odzyskania największej możliwej ilości danych po wystąpieniu błędu. Praca dotyczy głównie kodów Huffmana.

W danych zakodowanych kodem Huffmana błąd bitowy powoduje, że dekoder traci synchronizację z koderem. Błąd propaguje się, ponieważ granice słów kodowych widziane przez dekoder nie pokrywają się z oryginalnymi granicami. W przypadku większości kodów Huffmana dekoder w końcu zsynchronizuje się. Niemniej jednak nie istnieje a priori ograniczenie górne na liczbę niepoprawnie zdekodowanych symboli.

Praca wprowadza dwie metody ograniczania propagacji błędów w kodach Huffmana do co najwyżej L bitów, gdzie L jest parametrem. W jednej metodzie zakłada się, że dekoder zna numer bitu, który jest dekodowany, w drugiej pozycja dekodera może być nieznana. Metody są oparte na synchronizacji dekodera, który startuje od dowolnego bitu zakodowanych danych. Wykorzystują one immanentną tendencję kodów Huffmana do spontanicznej synchronizacji i nie wprowadzają żadnej redundancji jeśli taka synchronizacja zawsze zachodzi. Kolejna wprowadzona metoda ograniczania propagacji błędów może być użyta do szerokiej klasy kodów. Metody zostały zastosowane do równoległej dekompresji danych skompresowanych kodami Huffmana, a pomyślne testy zostały przeprowadzone na kompresji Jpeg. Dodatkowo przedstawiony został efektywny algorytm znajdowania poprawnego ułożenia słów kodowych przez dekoder zaczynający działanie w środku danych zakodowanych zwykłym kodem Huffmana.

Statystyczna synchronizacja kodów Huffmana jest związana z ciągami synchronizującymi — ciągami, które zawsze resynchronizują dekoder. Zostało pokazane, że znajdowanie ciągu synchronizującego dla kodu Huffmana jest równoważne ze znalezieniem ciągu synchronizującego pewnego automatu skończonego. Przedyskutowana została hipoteza Černý'ego dla tej klasy automatów. Wprowadzono ograniczenie górne na długość najkrótszego ciągu synchronizującego. Ograniczenie to jest poparte efektywnym algorytmem sprawdzającym czy dany kod ma ciąg synchronizujący, a w przypadku pozytywnym, konstruującym ciąg synchronizujący o długości mieszczącej się w podanym ograniczeniu. Zostały znalezione dwie klasy kodów Huffmana z długim najkrótszym ciągiem synchronizującym. Dla tych kodów policzona została dokładna długość takiego ciągu. Praca zawiera również dwa efektywne algorytmy znajdujące dla danego kodu wszystkie synchronizujące słowa kodowe, czyli słowa kodowe, które są ciągami synchronizującymi.

Słowa kluczowe: kod Huffmana, synchronizacja, opóźnienie synchronizacji, gwarantowana synchronizacja, ciąg synchronizujący, synchronizujące słowo kodowe, znacznik synchronizujący, odporność na błędy, silna synchronizacja.

Klasyfikacja tematyczna ACM: E.4, F.1.1.

Acknowledgements

The person without whom not only the dissertation would not have been finished but also it would not have even been started is undoubtely my advisor dr hab. Wojciech Plandowski, prof. UW. He introduced me to the topic of Huffman code synchronization and gave me the subject of my research. He was always available for discussing any result I came up with. He taught me that proofs must always be precise and any possible way of improvement should always be explored. Our discussions were always fruitful and led to many improvements of my intermediate results. Had it not been for prof. Plandowski, the results presented in the dissertation would have been much weaker. He was always able to spot incredible amount of mistakes, improving the quality of the dissertation. Prof. Plandowski helped me also with preparing publications with results of my research making the papers more precise and readable.

The dissertation would not have ever been created without the education I received. First of all, my parents always encouraged me to learning, but, luckily, they did not put too much pressure on me. My interests in mathematics was given to me by my primary school mathematics teacher, mgr Barbara Lipiec. Then, my interests in science was deepened in the secondary school by my mathematics teacher, mgr Henryk Frynas, and my physics teacher, mgr Jacek Orzechowski. Next, my education was continued at the University of Warsaw.

The dissertation was written while I was a Ph.D. student at the University of Warsaw. I am grateful to the dean's office, in particular to mgr inż. Anna Osmańska-Zych for help with all formalities related to being a Ph.D. student.

I am grateful to my friends, Sylwia Słotwińska-Karaś and Grzegorz Miłoś for help with improving the language of the dissertation.

I am also indebted to dr hab. Damian Niwinski, prof. UW, for pointing out the relation between Huffman code synchronization and synchronization of finite automata. This resulted in the research covered in Chapter 4.

During the preparation of the dissertation I was financially supported by a Ph.D. scholarship of the University of Warsaw in 2007-2008 and by the grant of the Polish Ministry of Science and Higher Education N N206 376134 (thanks to dr hab. Wojciech Plandowski, prof. UW, for the latter).

I attended a number of conferences and workshops, where I presented the results of my research described in the dissertation. My visit DCC'08 conference was financed by the Institute of Informatics, University of Warsaw (thanks to prof. dr hab. Krzysztof Diks). My visit to ITW'08 workshop was partially supported by the grant of the Polish Ministry of Science and Higher Education N 206 004 32/0806 (thanks

to prof. dr hab. Wojciech Rytter) and partially by the Conference Scholarship of Warsaw Scientific Society. My visit to FIT'08 workshop was financed by the grant of the Polish Ministry of Science and Higher Education N 206 004 32/0806 (thanks to prof. dr hab. Wojciech Rytter). My visit to MFCS'08 conference was financed by the grants of the Polish Ministry of Science and Higher Education N 206 004 32/0806 (thanks to prof. dr hab. Wojciech Rytter) and N N206 376134 (thanks to dr hab. Wojciech Plandowski, prof. UW).

The dissertation was prepared using LaTeX typesetting system. It was written in Eclipse editor using TeXlipse plugin and compiled with MiKTeX implementation of TeX. A number of LaTeXpackages were used, among them GASTEX for automata diagrams, ALGORITHM2E for algorithm pseudocode, and SYNTTREE for tree drawings. The title pages of the disseratation are based on the official template of the faculty, prepared by Jaroslaw Buczynski. The style for the dissertation was prepared by me with help from my collegues Robert Dąbrowski, Łukasz Kowalik and Michał Strojnowski.

The test programs were written in Java programming language using Eclipse platform, with help of LOG4J and JUNIT packages. Tests were preformed on my personal computer and on a computer financed by the grant of the Polish Ministry of Science and Higher Education N N206 376134 (thanks to dr hab. Wojciech Plandowski, prof. UW). The results were processed with OpenOffice.org open-source office software suite and GNUPLOT plotting utility.

Contents

Chapter 1

Introduction

1.1 Data compression

The goal of data compression is to convert an input data stream into one of a smaller size [62]. A data stream may be, for instance, a file with an image, video, text, source code, executable, etc., data transmitted through a network or an array in memory. The operation can then be reversed to reproduce the original data. The main gain of using data compression is the reduction of storage requirements. It can be directly evaluated in terms of the number of discs required to store data, the amount of memory needed for a program execution or the capacity of a music player needed to store a collection of music. Any of these can be expressed in money savings. In addition, data compression reduces time needed for transferring data, for instance, over a network or from a disk to main memory.

Salomon [61] enumerates many different types of data compression, only a few of them to be mentioned here. A *nonadaptive* compressor does not modify its operations, its parameters or its tables in response to the input data being compressed. In contrast, in *adaptive* methods the parameters of the encoder are modified, depending on the data read from the input stream. In *lossy* compression, it is allowed to lose some information to achieve smaller size of the result. Lossy compression is used intensively in image, audio and video compression. In these cases the distortion is either beyond human perception or may be tolerated. On the other hand, *lossless* compression reproduces the input stream accurately. This is crucial for executable files, for instance, where a single bit error may cause execution of an incorrect instruction by the processor and failure of a program. Similarly, in text files an erroneous character may change the meaning of a sentence. A popular technique of data compression is called *cascaded compression*. The output from one decoder is an input for another one, to produce an even smaller file.

The efficiency of data compression is characterized by the compression ratio, that is the relative size of the compressed data:

$$\text{compression ratio} = \frac{\text{size of the compressed output}}{\text{size of the uncompressed input}}. \tag{1.1}$$

We expect a compressor to achieve the compression ratio less than 1 for its typical

input streams. It should be noted that in lossless compression it is not possible to achieve the compression ratio less than 1 for all input streams. Therefore, every compression method is suited to a certain class of data.

Sayood [64] divides the process of data compression into *source modeling* and *coding*. The goal of source modeling is to extract the information about any *redundancy* in the data, that is any unnecessary information that can be removed, and to model it. Then, in the coding phase the information about the model and the information describing the data are written to the output stream. The latter information may be, for instance, a description of how the data differ from the model. As an example, in audio compression raw data is modeled as a signal consisting of different frequencies. The modeling phase transforms the time-domain signal to the frequency domain. Then, in the coding phase, the most important frequencies are written to the output stream using a certain code.

In many data compression algorithms it is assumed that the input stream, also called the *source* or the *source message*, consists of characters, called *symbols* or *letters*. Some methods investigate the correlation between symbols and existence of patterns in data and use them to reduce the compression ratio, as in case of LZ-like algorithms [64]. Other methods, called *entropy coders*, for example Huffman codes or arithmetic coding, assume that the symbols are independent and identically distributed random variables and use the knowledge about frequencies of symbols. In the latter case, the amount of information in each character of a source $\mathcal{S}$ is measured by its (binary) entropy, which is defined by:

$$H(\mathcal{S}) = -\sum_i p_i \log_2 p_i. \tag{1.2}$$

The sum is extended over all possible values of a character, and p_i is the probability that a symbol is equal to the i-th character. The entropy of the n-symbol stream is $nH(\mathcal{S})$ and it is a lower bound on the average size of a compressed stream[1] [64]. Entropy coders are often used in the final stage of cascaded compression schemes, after data is transformed into a new string with little correlation between symbols.

A popular class of data compressors is based on *variable-length codes* [62] and is used usually for entropy coding. In encoding, each symbol is replaced by some string, called the *codeword* for the symbol. The codewords may be of different lengths, hence the name: variable length codes. The set of codewords is designed in such a way that the decoder is able to replace codewords with their corresponding symbols to reconstruct the original stream. There are various procedures of choosing codewords for the symbols.

Some variations of variable-length codes do not follow strictly this way of encoding. For instance in *Tunstall codes* [62] each codeword corresponds to several symbols. In *redundancy feedback* codes [62] each symbol has several codewords. In the latter codes, only parts of codewords are written to the output stream.

Huffman codes [30] are variable length codes used for entropy coding. A certain probability distribution is assumed on the input characters. Usually the distribution

[1] Under some natural assumptions on the decoder.

is generated by counting occurrences of characters in the input stream. Codewords are assigned to symbols in such a way that the weighted average codeword length, $\sum_i p_i l_i$, is minimal. Here p_i is the probability of the i-th character and l_i the length of its codeword. Huffman codes are *prefix-free* (or *prefix*, for simplicity), which means that it is not possible for a codeword to be a prefix of another codeword. In this dissertation only binary Huffman codes are considered, i.e. codes where codewords are binary strings. Binary Huffman codes are *complete*, which means that any binary string longer than the longest codeword has some codeword as its prefix. As a consequence, it is not possible to detect an error in Huffman-encoded data before the end of the compressed stream.

Example 1.1: The following assignments of codewords to letters is a Huffman code, hereinafter called C_1:

$$\begin{array}{ll} a & \to 00 \\ b & \to 01 \\ c & \to 10 \\ d & \to 110 \\ e & \to 111 \end{array} \tag{1.3}$$

The code is prefix-free, because no 3-bit codeword starts with 00, 01 or 10. The code is complete because any binary string of length 4 has some codeword as its prefix. For instance, the string 0110 has the codeword 01 as a prefix.

The properties of being prefix-free and complete provide an easy way to decode Huffman codes. Consecutive bits of the encoded stream are read until they form a codeword. It will always happen, because the code is complete.

Huffman codes may be constructed using the Huffman algorithm [30]. The algorithm takes probabilities of symbols as input and produces optimal code for the given probability distribution. A clear description of the algorithm can be found in any textbook on data compression [62, 61, 64] and will not be repeated here.

Optimal complete prefix codes are not unique. In the Huffman algorithm there is always a possibility of interchanging the labels 0 and 1 when two parts of the code are joined (see e.g. [64]). Any choice will result in an optimal code. Moreover, for some probability distributions, there are optimal prefix codes that cannot be constructed using the Huffman algorithm [62, 22, 46]. In this dissertation any complete binary prefix code is called a Huffman code because it is an optimal code for some probability distribution.

Example 1.2: The following code, C_2, has codewords of the same length as code C_1 from Example 1.1. It means that messages encoded with codes C_1 and C_2 are of equal length.

$$\begin{array}{ll} a & \to 00 \\ b & \to 10 \\ c & \to 11 \\ d & \to 010 \\ e & \to 011 \end{array} \tag{1.4}$$

Huffman codes nearly reach the entropy of the input stream. The bound on the average length of an encoded character, $\bar{l}$, is [64]:

$$H(\mathcal{S}) \leq \bar{l} \leq H(\mathcal{S}) + p_{\max} + 0.086, \tag{1.5}$$

where $p_{\max}$ is the probability of the most probable symbol.

A more recent entropy coding method is *arithmetic coding* [64]. This is not a variable length code, but can rather be regarded as an assignment of a long codeword to the entire input [64]. Arithmetic coding reaches the entropy even closer:

$$H(\mathcal{S}) \leq \bar{l}_A \leq H(\mathcal{S}) + \tfrac{2}{m}, \tag{1.6}$$

where $\bar{l}_A$ is the average number of bits per source symbol and m is the number of symbols in the input stream. Nevertheless, because of the simplicity, robustness and fast decoding, Huffman codes are of a great importance in data compression.

1.2 Error resilience in compressed data

1.2.1 Error resilience

Error resilience is the ability of data to keep information in the presence of errors. Typically, the errors are the following:

- *bit-flip* errors — a single bit changes its value from 0 to 1 or from 1 to 0,
- *bit insertion* — an additional bit with an arbitrary value is inserted into data at some place,
- *bit deletion* — one bit is removed from data,
- *burst* error — a number of consecutive bits are corrupted, i.e. their value becomes random,
- *missing fragment* — a fragment of data is missing.

In channel coding theory [42] a certain probability of errors is assumed. The information may be encoded using *error correcting codes*. These codes add some redundancy to data in such a way that the probability of a bit error after decoding is less than a certain threshold.

Example 1.3: Consider encoding each bit of the source as three bits of the same value. For the bit 0 the encoding is 000 and for the bit 1, it is 111. The encoding for the message 0110 is 000 111 111 000 (spaces are inserted just for readability). Now, let us consider a bit-flip error in the fifth bit of the encoded message. The result is 000 101 111 000. The first, the third and the fourth triple can be decoded normally as 0, 1 and 0, respectively. The second triple, 101, corresponds to neither 0 nor 1. This is an indication that a bit error has occurred. If the bit errors are independent and occur with a small probability,

it is more likely that the string 101 is the result of flipping the second bit of 111 than flipping the first and third bit of 000. The string 101 is decoded as 1 and the whole string is decoded correctly as 0110.

The redundancy of the code is 2, which means that there are two additional bits per each coded bit. The code is resistant to at most one bit-flip error within a codeword. It should be noted that there are error correcting codes with much smaller redundancy and better error resilience.

Error correcting codes may be regarded as the opposite of data compression as the latter removes redundancy. Compressed data is susceptible to bit errors because there is no redundancy that could help to correct them. What is even worse, a single bit error in compressed data may cause more errors after decompression. The data read by the decoder influences the decoder's state and, being in a corrupted state, the decoder may misinterpret the remaining part of the data. The error will thus propagate even till the end of the decoded stream.

As pointed out by Wen and Villasenor [72], in typical applications data already contains error correcting codes. If an error is detected by these codes, the data is retransmitted (or reread from disk). Nevertheless, in delay-constrained applications, for example in audio or video streaming, retransmissions become problematic. Then, the issue of how to use compressed data in error-prone environments becomes vital. The goal is to ensure that after a bit error, even though some data is lost, as much data as possible is recovered.

1.2.2 Error resilience of variable length codes

The following example reveals the influence of bit errors on data encoded with variable length codes. It is assumed that the code has the prefix property, i.e. no codeword is a prefix of another codeword.

Example 1.4: Huffman code C_1 from Example 1.1 is considered below. Let the source message be

$$\mathcal{M} = \text{`}bbeaaebcec\text{'}. \tag{1.7}$$

Let $C_1(\mathcal{M})$ be the result of encoding this message with the code C_1. Assume that the third bit of the encoded message was flipped, giving an erroneous message, $C_1(\mathcal{M})_e$. The division into codewords for $C_1(\mathcal{M})$ and $C_1(\mathcal{M})_e$ is shown below. The flipped bit is marked in bold.

$$C_1(\mathcal{M}) = \underbrace{0\,1}_{b}\ \underbrace{0\,1}_{b}\ \underbrace{1\,1\,1}_{e}\ \underbrace{0\,0}_{a}\ \underbrace{0\,0}_{a}\ \underbrace{1\,1\,1}_{e}\ \underbrace{0\,1}_{b}\ \underbrace{1\,0}_{c}\ \underbrace{1\,1\,1}_{e}\ \underbrace{1\,0}_{c} \tag{1.8}$$

$$C_1(\mathcal{M})_e = \underbrace{0\,1}_{b}\ \underbrace{\mathbf{1}\,1\,1}_{e}\ \underbrace{1\,1\,0}_{d}\ \underbrace{0\,0}_{a}\ \underbrace{0\,1}_{b}\ \underbrace{1\,1\,0}_{d}\ \underbrace{1\,1\,0}_{d}\ \underbrace{1\,1\,1}_{e}\ \underbrace{1\,0}_{c} \tag{1.9}$$

The decoding of the erroneous message is *bedabddec*. The first letter, *b*, was decoded correctly. The error occurred in the second codeword and the second letter was decoded as *e*, instead of *b*. The next letter is also incorrect and the error propagates until the last two letters.

We can see that even a single bit error propagates because the codewords read by the decoder after the error are not aligned with the original codewords. The bits that form a decoded codeword of the erroneous string may be taken from a suffix of some real codeword and a prefix of the next one. For instance, the third codeword, 110, was formed of a suffix of the codeword 111 and a prefix of the codeword 00. In such case the decoder is called *unsynchronized*.

In Example 1.4 the decoder is unsynchronized from the third to the seventeenth bit. It is interesting that some number of bits after the error the decoder *resynchronizes* [22, 46], i.e. the codeword boundaries it uses are correct again. In Example 1.4 it happens after the eighteenth bit. From then on, the decoder decodes correct data again, which means that such decoder is *synchronized* afterwards.

The loss of synchronization after a bit error is equivalent to the behavior of a decoder that starts at some arbitrary bit in the encoded stream. For instance, the codewords of the erroneous message are the same as those decoded by a decoder that started at the sixth bit of the error-free message. This property will often be used in this dissertation.

The resynchronization seen in Example 1.4 is not a coincidence, but a general property of Huffman codes. It is the main subject discussed in the dissertation. There are various techniques that improve synchronization properties of Huffman codes. The aim of these synchronization schemes, as pointed out by Perkins and Smith [52], is not to recover all of the data but rather to ensure that after the loss of some data due to error, the decoding of subsequent data is correct. Additionally, for some applications it is important to deduce the position of the recovered data in the data stream, which is called *strong synchronization*.

1.2.3 Synchronizing strings

Many Huffman codes have a *synchronizing string* — a string that always resynchronizes a decoder that processes it. Such strings can be defined as follows:

$$s \text{ is a synchronizing string } \Leftrightarrow \begin{array}{l}\text{for any bit string } w\\ \text{the string } ws \text{ is a sequence of codewords}\end{array} \tag{1.10}$$

Example 1.5: The string 0110 is a synchronizing string for the code C_1 from Example 1.1. To see that, it is enough to check the right hand side of (1.10) for all the strings w that are proper prefixes of codewords (other strings have a codeword as a prefix), that is for the empty string, 0, 1, and 11. The division of the strings ws (w followed by s) into codewords are 01 10, 00 110, 10 110 and 110 110, respectively, so these are sequences of codewords.

By taking $s = \epsilon$ (the empty string) in (1.10) it is vivid that a synchronizing string is a sequence of codewords.

The distribution of codewords' lengths for a code with a synchronizing string was analyzed by Schützenberger [65]. It is obvious that if the greatest common divisor (GCD) of codewords' lengths is greater than 1, no synchronizing string for the code exists. Indeed, if s was such a synchronizing string, and $d > 1$ was the GCD, the sequence $1s$ (the concatenation of 1 and s) should be a sequence of codewords. But this

is not possible since the length of the string $1s$ is not divisible by d. Schützenberger proved that if GCD of codewords' lengths is 1, there exists a Huffman code with codewords of the same lengths that has a synchronizing string. He also gave a method for the construction of such a code. It should be noted that GCD equal 1 does not imply that a particular code has a synchronizing string.

Berstel and Perrin [5] proved that if GCD of lengths of the codewords 0^i and 1^j, formed of all zeros and all ones, is 1 then the code has a synchronizing string. Their book includes also a chapter devoted to, so called, *biprefix* codes. Such codes satisfy the prefix condition, and also the suffix condition: no codeword is a suffix of another codeword. It is easy to prove that such codes cannot have a synchronizing string.

Various authors considered the construction of codes with a short synchronizing string. Rudner [60] gave a method for constructing codes with the shortest synchronizing string possible if the shortest codeword is of length $m = 1, 2, 3, 4$. Although the author does not state it explicitly, for $m > 1$ such a synchronizing string must be a codeword, which is interesting itself. The work of Rudner has been later extended, for instance in [19].

Capocelli et al. [12] analyzed *statistical synchronizability* of Huffman codes. A code is statistically synchronizable if the probability of resynchronization tends to 1 with the number of decoded symbols going to infinity. They proved that a variable-length code is statistically synchronizable if and only if it has a synchronizing string (under the assumption of *ϵ-guaranteed message source*, see [12]). They also gave an algorithm to test whether a code has a synchronizing string. The algorithm works for all variable-length codes and an optimized version for prefix codes has complexity $O(N \sum |w_i|)$, where w_i are codewords. (This dissertation contains an algorithm of complexity $O(\sum |w_i|)$.)

Codes that have a synchronizing string are not rare. Freiling et al. [24] proved that almost all Huffman codes have a synchronizing string. Precisely, if we choose a Huffman code with N codewords at random, the probability that it has a synchronizing string goes to one with N going to infinity.

There are, however, such distributions of codewords' lengths that no code with a synchronizing string exists. These are the distributions with GCD of codewords' lengths grater than one. Capocelli et al. [13] proposed such a construction of suboptimal codes with a synchronizing string that the average codeword length is increased by at most $p_{\min}$ — the probability of the least probable symbol. The method first modifies codewords' lengths in order to get two codewords with GCD of their lengths equal 1. Then, the code is rearranged so that these codewords are 0^i and 1^j. By the result of Berstel and Perrin [5] such a code has a synchronizing string. The article also describes a method for constructing suboptimal codes with a synchronizing string that is a codeword (see also the next section).

Some properties of minimum redundancy codes with a synchronizing string were also collected by Long et al. [41].

1.2.4 Synchronizing codewords

It is rather surprising that a synchronizing string may also be a codeword. In this case, it is called a *synchronizing codeword*. The study of synchronizing codewords was pioneered Ferguson and Rabinowitz [22], however, before that, Rudner [60] presented an algorithm for the construction of codes with a synchronizing string that, in fact, was a codeword. Rudner proved that if the length of the shortest codeword is $m > 1$, the length of the shortest possible synchronizing string is at least $m + 1$. The algorithm of Rudner constructed codes with a synchronizing string of length $m + 1$, for $m = 2, 3, 4$, under some additional assumptions. Even though the article did not mentioned it, such a synchronizing string must be a codeword. Indeed, it is a sequence of codewords and there is no codeword of length 1 nor of length smaller than m.

If the probability of a codeword is p, the codeword appears, on average, at each $\frac{1}{p}$-th symbol. The existence of a synchronizing codeword in the code would limit the *synchronization delay*, that is the number of symbols lost before resynchronization, to $\frac{1}{p}$, on average. If there are more synchronizing codewords, with probabilities p_1, $\ldots p_k$, the average synchronization delay can be approximated by $(\sum_i p_i)^{-1}$. This is not a precise expression, since the synchronizing codewords may also appear as suffixes of other codewords. In fact, the average synchronization delay is even lower. This motivation for the research related to synchronizing codewords was presented by Ferguson and Rabinowitz [22]. A more detailed analysis of the average synchronization delay for codes is given in [46].

The work of Ferguson and Rabinowitz [22] focused entirely on synchronizing codewords. They gave a number of necessary and sufficient conditions for a code to have a synchronizing codeword. Simultaneously, they presented a method for the construction of an optimal code with a synchronizing codeword for some distributions of codewords' lengths. The method was applied to construct codes with a synchronizing codeword for the probability distributions of letters in English and French languages.

Montgomery and Abrams [48] came up with an algorithm for the construction of a suboptimal code with a synchronizing codeword. The redundancy is added by extending the longest codeword by one bit. Their algorithm, as the one of Ferguson and Rabinowitz [22], works only for some probability distributions.

Capocelli et al. [13] showed how to transform a code to a suboptimal one with a synchronizing codeword. The redundancy per symbol is increased by at most $\frac{1}{d}$, where d is the size of the target alphabet, 2 for binary codes. Their code is not complete and in this case the definition of a synchronizing string is slightly different. The reader is referred to [13] for more information.

Escott and Perkins [19] considered binary Huffman codes whose shortest codeword is of length $m > 1$ and that contain a synchronizing codeword of length $m + 1$, the shortest possible in this case. This was the continuation of Rudner's [60] work, that extended it in several cases. The authors created an algorithm for constructing such codes for a given set of codeword lengths, provided that such code exist (in contrast, Rudner's algorithm works only for $m = 1, 2, 3, 4$). They also considered the existence of synchronizing codewords of other lengths in these codes and proved that for $m \geq 3$ there must be at least one more codeword that is synchronizing. The

authors discussed synchronization properties of their codes, measured, for instance, in terms of the expected frequency of finding a synchronizing codeword (as in [22]).

Another approach was taken by Perkins and Escott in [51]. In contrast to their previous work [19], they assumed that a certain string w is a synchronizing codeword and from the properties of w they inferred some properties of the code. They described the relationship between the length of the shortest codeword in a Huffman code and the length and the structure of the synchronizing codeword. The analysis was not restricted to binary codes, but complete prefix codes of any arity were considered as well. The authors presented an algorithm that builds a code with a given string as a synchronizing codeword (but the code is not built to follow a given probability distribution on symbols). A tight upper bound on the length of the shortest codeword in such a code was given, which led to an interesting result that if the length of the shortest codeword, m, and of the longest codeword, M, are related by $M < 2m - 1$ then the code cannot have a synchronizing codeword.

Huang and Wu [29] extended both the work of Rudner [60] and of the work of Escott and Perkins [19]. The authors proved that if the length of the shortest synchronizing codeword is $m + 1$, where $m > 1$ is the length of the shortest codeword, the shortest synchronizing codeword is either 01...11 of 01...10 or both (this is a restatement of the result originally given by Rudner [60]). They also gave necessary conditions for the existence of a binary Huffman code with such shortest synchronizing codeword(s). Finally, they presented an algorithm for the construction of such codes provided that all the necessary conditions have been met.

1.2.5 Average synchronization delay

Given a number of optimal codes for a certain probability distribution on symbols it is important to be able to find the one with the best synchronization capabilities. To do that, a good measure of such synchronization capabilities of codes is needed. Ferguson and Rabinowitz [22] argued that the sum of probabilities of symbols whose codewords are synchronizing reflects the tendency of the code to resynchronize. The authors stressed that this is only an approximation. This approximation is useless, however, if the code does not have a synchronizing codeword.

Maxted and Robinson [46] investigated the problem more profoundly. They used a state diagram for the computation of the average synchronization delay for a code. The average is taken over the probability distribution of source symbols. They assumed that the letters are statistically independent identically distributed random variables. The states of the diagram are prefixes of codewords and correspond to the prefixes of codewords already read by a decoder when it is on an actual codeword boundary. A synchronized decoder always sees the correct codeword boundaries so the prefix is always the empty word. As finding the expected synchronization delay is computational expensive, the authors also gave a simplified approximate method.

As a result of testing the average synchronization delays of codes, Maxted and Robinson [46] identified two properties of codes that are important for good synchronization capabilities. The first one is that the code contains many codewords that differ by only one bit. In such a case a bit-flip error often does not cause a synchroni-

zation loss. The other property is that there are many codewords that are suffixes of other codewords. This allows for quick resynchronization after synchronization loss. The authors stress that the second property seems to be more important. The article also contains results of simulations of resynchronization after an error.

The work of Maxted and Robinson [46] was an inspiration for many other researchers. The first extension, due to Monaco and Lawler [47], corrected a few minor (mostly computational) mistakes in [46]. It also introduced a method for the calculation of the variance of the expected synchronization delay. Other articles [66, 68, 67, 74], to be described shortly, simplified the way of computing the average synchronization delay and its variance. Several methods for constructing codes with good synchronization potential have been proposed [74, 73, 69].

Earlier, Rahman and Misbahuddin [57] considered transmission of a Huffman-encoded message over a binary symmetric channel (BSC) with crossover probability p (bit-flip errors occur independently, with probability p). They used a state diagram similar to the one of Maxted and Robinson, but this time they included the possibility of several errors in one codeword and the possibility of errors during the error recovery process, before resynchronization occurs. The authors provided an approximation for the calculation of the expected synchronization delay that seems to be very close to the exact value, although no bound on the error was given. Also the computational complexity of the method was not analyzed. The authors considered only bit-flip errors, but the method can easily be extended to analyze other types of errors. Additionally, they analyzed the influence of the BSC crossover probability, p, on error recovery process and on the synchronization delay. In some cases the delay is an increasing function of p, in other cases — decreasing. They also investigated the dependence on p of the expected total number of symbols lost.

Al Soualhi and Hassan [66] improved the work of Maxted and Robinson [46]. They showed another approximate method for the calculation of the expected synchronization delay and its variance. Their method can be made as close to the exact solution as needed.

Takishima et al. [68] analyzed error recovery of Huffman codes after a bit-flip error. The authors, like Rahman and Misbahuddin [57] and unlike Maxted and Robinson [46], also considered possible errors during the recovery phase. They gave a matrix expression for the average synchronization delay. The authors discussed shortly the measure of code synchronizability given by Ferguson and Rabinowitz [22]: the sum of probabilities of synchronizing codewords. They gave an example of two codes, where the code with larger probability of a synchronizing codeword has worse average synchronization delay.

Takishima et al. [68] proposed an algorithm for rearranging the code to increase the probability of resynchronization. It is based on the idea of the code being "suffix rich" — having many codewords with other codewords as suffixes. This is the second criterion of Maxted and Robinson [46]. The algorithm works as follows: the shortest codeword is considered a seed, then other codewords are lined from the shortest ones and their structure, whenever possible, is transformed in such a way that they end with the seed. Then, all possible transformations of the seed, i.e. rearrangements of the code so that the seed is any string of the fixed length, are tested. After that, the

next shortest codeword is taken as the seed. Finally, the chosen code is the one with the largest number of suffixes of codewords that equal the seed.

Swaszek and DiCicco [67] gave an algebraic formulation and an exact solution for the problem of computing the expected synchronization delay and its variance. The analysis is reduced to simple operation on matrices: inversion and multiplication and the method can be used with software for symbolic computations, such as Maple. The authors analyzed the average numbers of symbols lost due to a single bit error. They proved that the probability of a character error is approximately equal to $Lp\mu$, where L is the average codeword length, μ is the mean error propagation length and p is the probability of a bit-flip error (p is small). Thus, if the inverse of the compression ratio is larger than the mean error propagation length, variable length encoding gives less character errors than fixed length encoding.

Zhou and Zhang [74] derived a simple equation for the average synchronization delay (called MEPL for mean error propagation length) and its variance (VEPL). They also proposed two heuristic algorithms for finding Huffman codes with low MEPL and VEPL. Performance comparison with other methods showed the superiority of their heuristics (see also Sections 3.8.3 and 3.8.4).

Yang and Kumar [73] developed a method similar to one of the methods of Zhou and Zhang [74]. The method is based on Rudner's claim that the codes should be suffix rich. Titchener [69] constructed codes that exhibit low synchronization delay. He also estimated the expected synchronization delay for his codes. These codes only exist for some probability distributions.

In their work on parallel Huffman decoding, Klein and Wiseman [38] provided yet another measure of codes' synchronization capabilities. It is based on counting codewords' suffixes that are sequences of codewords. They did not give any estimate of how their measure compares to the expected synchronization delay. They argued that canonical Huffman codes [35] have good synchronization capabilities. This is obviously wrong in view of the results presented in this dissertation, in Section 3.8.

1.2.6 Strong synchronization

Even though the decoder of a Huffman code eventually resynchronizes in most cases, the number of symbols it has decoded may be incorrect, as pointed out by Swaszek and DiCicco [67]. The error further propagates, because following symbols are placed at wrong positions in the decoded message. Many application rely on the positions of bytes in the stream. Consider, for instance, a set or records in a file where the first byte of each record holds its size. If there is a slippage in decoded data, the software interprets some internal bytes of records as records' sizes, which makes no sense.

Swaszek and DiCicco [67] developed a method for computing the probability of any number of extra characters decoded due to errors. For comma codes (codes of the form $1, 01, 001, \ldots, 000\ldots01, 000\ldots00$), for instance, it is almost always one symbol more or one symbol less. Calculations for another code can be found in [44]. In this case the probability that after a bit error the decoder reads one symbol less is 10%, one more — 6% and the probability of decoding the correct number of symbols is 84%.

The problem of correcting symbols' positions was investigated in a number of papers. The ability of a decoder to not only resynchronize but also to place the following symbols correctly is called *strong synchronization* or *symbol synchronization*, and can be achieved by inserting additional markers into encoded message.

Lam and Kulkarni [40] introduced the concept of an *extended synchronizing codeword* (ESC). After a decoder receives an ESC, it correctly knows that such a codeword has been received (normal synchronizing codeword may be a suffix of other codewords, so it may be received implicitly) and it may invoke an error concealment procedure. The authors described a method for the construction of the ESC, done by first rearranging the code and then by extending the longest codeword by a few bits, so that the ESC is not a substring of any concatenation of other codewords. The authors computed the required number of bits that need to be appended to the longest codeword to form the ESC, depending on the lengths of other codewords. They also discussed the redundancy of codes with an ESC.

The authors pointed a number of ways for providing the decoder with strong synchronization using an ESC. They proposed regular insertions of the ESC in the encoded stream. In this case, provided that the ESCs themselves are not corrupted, the decoder always knows the current symbol number after being resynchronized with an ESC. Another method is to insert the ESC followed by a counter. The counter gives the number of the current symbol or the number of the inserted ESC.

The construction of an ESC was generalized by Perkins and Escott [55]. They allowed the ESC to be created by extending an arbitrary codeword in the code, not only the longest one, as in [40]. This gives the possibility to create an ESC of a given length. The length of the ESC should correspond to the expected frequency of its insertions. The authors gave an exact lower bound on the length of the shortest ESC created by extending a given codeword, and presented an algorithm that achieves this bound.

The problem of providing the decoder with strong synchronization by inserting counters, outlined in [40], was discussed in depth by Perkins and Smith [52]. The authors analyzed a general scheme for strong synchronization. It is based on inserting a number of distinct keywords into the message, at intervals. The keywords consist of a synchronizing sequence and an explicit or implicit cyclic counter. The number of encoded symbols between consecutive keywords, K, is constant. There are N distinct keywords and they are inserted in fixed, cyclic order. Thus, if decoding of all the keywords is successful, strong synchronization is achieved. Decoding a keyword only gives the position in the stream modulo NK. An error in any keyword may cause a slippage of $\pm N$ keywords, that is $\pm NK$ characters. The paper of Perkins and Smith contains two algorithms for dealing with errors in keywords. The algorithms aim to minimize the probability of strong synchronization loss while keeping the required buffer size for the data low.

Perkins, et al. [54] presented a comparison of a number of strategies for strong synchronization. They focused on environments with reasonably high error rates and aimed to avoid permanent loss of strong synchronization when very large sets of data are transmitted or stored. The strategies are based on inserting a synchronizing keyword with a counter (explicit or implicit) after each N-th encoded symbol. The

authors do not assume that the synchronizing keywords do not appear in the encoded data. In case of such an occurrence, the string is falsely identified as a synchronizing keyword and therefore causes an error, that may or may not be corrected. Such errors appear even if no bit errors occurred in the data. The techniques analyzed in the paper deal with such errors, and with errors caused by a flipped bit as well. The authors compared the method with *bitstuffing*.

Bitstuffing is a technique that prevents occurrences of certain patterns in data. It will be explained using the following example. Assume that the occurrences of the string 111 in the message should be avoided. If 111 appears in the message, it is encoded as 1101. On the other hand, if 110 appears, it will be replaced with 1100. In such a way, the original message can always be recovered and the string 111 does not appear in the new message.

Perkins, et al. [54] also analyzed the combination of their schemes with bitstuffing. Bitstuffing prevents errors caused by an occurrence of a synchronization keyword in the original data. The advantage of the schemes presented in the paper is that they impose few restrictions on the form of data compression used, so they may be combined with many compression techniques. Perkins and Smith [53] applied the same analysis to burst errors.

Kashyap [34] described a similar method for providing strong synchronization. He proposed to insert a marker after every L-th codeword. The marker is 111..1 followed by a fixed-size integer — the marker number. Each sequence of L codewords is *bitstuffed* so that the string 111..1 does not appear in it. Such a code is a special case of the codes analyzed in [54]. The aim of Kashyap's work was to maximize the range of marker numbers for a given bit rate. The author gave the optimal length of 111..1 and the optimal value of L to achieve the largest range of marker numbers.

Fang and Jeong [21] analyzed the expected amount of data lost due to errors. In their setting the message is divided into fixed-length packets with a synchronization marker at the end. The authors assumed that if an error occurs inside a packet, all the data from the erroneous bit to the end of the packet is lost. They found the packet length that maximizes the number of bits correctly received. They also analyzed reversible codes, that allow for decoding backwards (for instance biprefix Huffman codes, see [23]). In this case, if the two resynchronization markers that delimit the packet are free of errors, the bits from the beginning of the packet to the first error and from the last error to the end of the packet are correct.

1.2.7 Application of error resilient variable length codes

There are several articles concerning application of synchronization and strong synchronization to reduce the influence of errors. For instance Lam and Reibman [39] applied extended synchronizing codewords to limit error propagation in Jpeg [64] images (actually this work preceded the work of Lam and Kulkarni [40], where ESCs were presented closer). The ESC were placed in fixed position to control error propagation due to synchronization slippage. The authors also developed an error concealment procedure to further improve the quality of images with errors.

Hemami [28] applied the methods for synchronization of Huffman codes to Jpeg

images and wavelet-coded images [64]. In her method, first a code with a synchronizing codeword is created using the algorithm of Rudner [60]. If the constraints for codeword lengths that are prerequisites for Rudner's algorithm are not met, the code is changed to a suboptimal one (for Jpeg images this is done by adjusting the quality factor — the parameter that measures information loss during quantization). Then, an extended synchronizing codeword is added to the code. The ESC is used as the end-of-line symbol that replaces the end-of-block symbol at the last block of each line. The existence of a synchronizing codeword in the code implies that the code resynchronizes quickly. In addition, the ESC provides robust positional information.

Yang and Kumar [73] considered Huffman encoding of subband coefficients in a wavelet-coded image and analyzed error detection and recovery. An error is detected if the decoder receives a wrong number of symbols. Errors are repaired using inter-subband correlation. The correlation coefficients are sent to the decoder together with the message.

A survey of another approach to dealing with errors in image and video transmission was presented by Kang and Leou [31]. In this approach, additional pieces of data are embedded in images or videos in order to use it in the error concealment procedure. The embedded data is transparent to applications that do not use it, so the schemes are compatible with existing software. The price for embedding additional data is the lowered image or video quality, but the loss in quality should be almost invisible for a human. In Jpeg images [64] the data may be, for instance, embedded in the least significant bit of a quantized DCT coefficient. The information used for error concealment may be, for instance, a downscaled version of the original image or a description of similarities between parts of the image. The data is embedded in a way that minimizes the probability of the loss of both a fragment of original data and its error-correcting data.

Klein and Wiseman [38] developed a parallel version for the decoder of Huffman codes. Each processor decodes its own fragment and also stores the position of the first bit of each codeword. Of course, a processor may be unsynchronized at start, but in most cases it resynchronizes quickly. After processor p finishes its block, it decodes a prefix of the next fragment, the one that belongs to processor $p+1$, until the codeword boundaries stored by processor $p+1$ indicate that processor $p+1$ is in synchronization with processor p. In case processor $p+1$ does not synchronize until the end of its block, processor p continues in block $p+2$, and so on. In the worst case, the whole message is decoded by the first processor, but this is extremely rare for typical data.

The method of parallel Huffman decoding is applied in [38] to Jpeg [64] images with good results. The most important problems with Jpeg images are that the decoder does not know where to place the decoded symbols (no strong synchronization) and that the DC coefficient are coded using the DPCM technique (Differential Pulse Coding Modulation: only differences from the previous value are stored [64]). The first one is solved by placing the parts of the image at arbitrary positions and moving them later. The other is solved by setting an arbitrary brightness by each processor at start and readjusting it later.

Klein and Shapira [37] analyzed pattern matching directly in Huffman compressed

messages. The pattern is first Huffman-encoded and then the result is matched to the compressed data. This may result in many false-positives when the codewords of the pattern are not aligned with codewords of the message. The authors present estimations of the expected number of false positives for given Huffman code, message and pattern.

The number of false-positives in the search is then reduced by using self-synchronization of Huffman codes. After a pattern has been found at index i of the encoded stream, the algorithm jumps back by a constant, K, number of bits and starts decoding from there. If K is chosen large, it is highly probable that the decoder resynchronizes before position i. It can be then decided, with a small probability of a false-negative or a false-positive error, whether i corresponds to a real occurrence of the pattern. The authors pointed out that because the decision whether a match is correct is probabilistic, the best choice for the algorithm for matching the encoded pattern to the encoded message is Karp and Rabin's probabilistic pattern matching [33].

1.2.8 Other methods for error resilient coding

Self synchronization is not the only way to improve the error resilience of Huffman codes. Fraenkel, Klein [23] proposed decoding of Huffman-encoded messages both from start and from the end. This increases error resilience, in particular a single bit error, if located properly, only influences the codeword where it appeared. Huffman codes cannot be easily decoded from the end, because one codeword may be a suffix of another one (the suffix property does not hold). The authors proposed the usage of biprefix [5] codes (called *bidirectional* in [23]) to facilitate the decoding. Additionally, they gave an algorithm to decode normal Huffman codes backwards. In most cases such decoding does not require much overhead. It should be noted that biprefix codes are not self-synchronizing. In fact, if a decoder loses synchronization, it will not resynchronize unless there is another error that puts it back in synchronization.

Wen and Villasenor [72] presented a construction of biprefix codes (called *reversible* variable length codes there) equivalent to the Rice-Golomb codes [26, 64] and exponential Rice-Golomb codes [62, 72]. They performed an analysis of error detecting features of the codes. Biprefix codes can detect more errors, because they do not resynchronize when synchronization is lost. The indication of an error is when at the end of the data stream there remain bits that do not form a complete codeword. Not all errors are detected because some bit errors do not cause a loss of synchronization. Also another error may resynchronize the decoder. On the other hand, the synchronization delay is larger, because they may only resynchronize after another error. As a consequence, more data is lost. The codes described in their paper have been adopted into ITU H.263+ video coding standard and served as the subject of an MPEG-4 core experiment.

Neuman [50] considered variable length codes defined by finite automata with output (Mealy machines [63]). The end of a codeword is defined by the occurrence of a particular output value of the automaton to which the encoded message is presented as input. For finite codes the construction always gives a complete binary prefix code,

but the author focused on infinite codes. He identified a special class of such machines, for which the output is a function of only the last encoded bit and the previous N bits, where N is finite. Neuman showed that such characteristics implies that a bit error destroys $N - l_{\min} + 2$ codewords at most (if only the value is greater than zero), where $l_{\min}$ is the length of the shortest codeword. This is done without any redundancy, but it should be kept in mind that the code is infinite. The author also proposed a method for inserting redundancy that allows for error detection. Finally, he investigated the problem of the construction of such a code for given symbols' probabilities.

Even [20] considered suboptimal variable length codes. He defined synchronizable codes of order N as codes for which the knowledge of the last N bits suffices to determine the correct codewords alignment. He gave necessary condition for a code to be synchronizable of a finite order and described an efficient algorithm to test it. The method was formulated in as a graph problem and it can also be applied to finite automata.

Another class of codes that synchronize quickly are *prefix-synchronized* codes, proposed by Gilbert [25]. These are fixed length codes with codewords of length N. A prefix of length $A < N$ is shared by all the codewords. It is called a *synchronizing prefix*. The remaining bits of the codewords are chosen in such a way that the synchronizing prefix does not appear in any other place in any sequence of codewords. This means that for a synchronizing prefix $P = p_1 p_2 \dots p_A$ the codewords are such sequences of bits $p_1 \dots p_A x_1 \dots x_{N-A}$ that P is not a substring of $p_2 \dots p_A x_1 \dots x_{N-A} p_1 \dots p_{A-1}$. The prefix serves as a synchronization marker for the decoder. Gilbert studied the number of possible codewords when N and A vary. For fixed A, the prefix $11 \dots 1$ is asymptotically the best. On the other hand, if N is fixed, it is always better to choose the one bit longer prefix $11 \dots 10$. It is conjectured that for fixed N the prefix $1^k 0$, with suitably chosen k is always the best in terms of the number of codewords in the code. The number of codewords in this case is roughly $0.35 N^{-1} 2^N$, which gives the redundancy less than $\log N + 1.52$ per codeword.

Guibas and Odlyzko [27] proved that the prefix-synchronized codes with $11 \dots 10$ as the prefix are optimal in terms of size of the code, for sufficiently large N (codeword length) for alphabet size equal 2, 3 and 4. They showed that the above conjecture is false for alphabet size greater than 4. The prefixes s that maximize the size of the code are the ones that are not self-correlated, i.e. no prefix of the string s is its suffix.

Morita et al. [49] described a procedure of mapping data sequences into codewords of a prefix-synchronized code (see Gilbert [25]) as well as the inverse mapping. They considered only maximal codes. Their algorithm works primarily for the prefix $11 \dots 10$ and is extended for any prefixes that are not self-correlated. It has been proved in [27] that such prefixes give codes of the same size as for the prefix $11 \dots 10$. The procedure does not require any lookup tables and the complexity of entire mapping is proportional to the length of codewords, so the decoding with the proposed method is efficient.

Yet another method of coding resistant to bit errors is called EREC, for Error Resilient Entropy Code, and was introduced by Redmill and Kingsbury [58]. This is a general method for adapting existing schemes to increase resilience to random and burst errors while maintaining high compression ratio. The method divides the data

into variable-length blocks in such a way that bit errors do not cause synchronization loss. It is assumed that the decoder knows when it finishes decoding a block. The variable-length blocks are put into fixed-length slots of size equal to the average block length. The part of a block that does not fit inside one slot is put into some other slot that has some space left. After a few iterations all the data is put into the slots. The slots are used in a fixed order so the decoder always knows where to find the remaining data. The method is applied to image and video compression schemes.

Malinowski et al. [44] analyzed how the information about the length of the encoded message may improve the error resilience of a variable length code. They investigated soft decoding with length constraint — trellis decoding technique based on *maximum a posteriori* method [42]. The above-mentioned length constraint is used to identify all decoded sequences having the number of symbols that differs from the number of transmitted symbols. Let ΔS be a random variable that describes the number of additional symbols decoded due to an error. It was proved in [44] that with soft decoding, in order to choose the best code, it is better to consider the probability of receiving an incorrect number of symbols, $\mathbb{P}(\Delta S \neq 0)$, and the entropy of the random variable ΔS than to consider mean error propagation length and its variance.

Malinowski et al. [43] used a state model, analogous to the one of Maxted and Robinson [46], to compute the average error propagation and its variance for quasi-arithmetic codes.

De Moura et al. [16] described a version of Huffman codes where codewords assigned to input symbols are sequences of whole bytes. The target alphabet of the Huffman code is of size 128 and the remaining bit in each byte is used as a tag to signal the beginning of a codeword. These codes are called *tagged* Huffman codes. Resynchronization with these codes is simple, because the decoder can always find the beginning of a codeword by examining the tag. The authors applied the codes to a fast compression and decompression scheme for natural language texts. The scheme allowed for exact search for words and phrases in compressed data. What is interesting, the authors claim that the search in such compressed data is faster than in uncompressed text. It is due to the fact that compressed text is shorter.

The tagged Huffman codes were replaced by *end-tagged dense codes* in [11] and by (s, c)-*dense codes* in [10]. Both of these codes consist of fixed codewords and do not depend on the probability distribution on source letters. They are better than tagged Huffman codes in terms of redundancy and provide the same synchronization properties.

Similar property of marked beginning of a codeword is present in *Fibonacci* codes [3]. The codewords in Fibonacci codes are also fixed. They are related to the representation of integers as a sum of Fibonacci numbers. In the code of order m the string of 1 consecutive ones may only appear as a suffix of a codeword. This property helps to resynchronize the decoder after synchronization loss. The robustness and other properties of Fibonacci codes were analyzed by Klein and Ben-Nissanin in [36].

1.2.9 Huffman coding and arithmetic coding

Huffman codes have become very popular, mainly because of their simplicity and little computational requirements. However, these codes do not always offer the best compression ratio, losing with arithmetic coding. It may appear that Huffman codes are thus obsolete.

Bookstein and Klein [9] presented a comparison between Huffman codes and arithmetic coding. They argued that for many applications Huffman codes are still a better choice. The authors analyzed the influence of dividing the code into blocks and also the influence of the End-Of-Block symbol on the redundancy of arithmetic codes. The division into blocks is important for separate decoding of data fragments and for error resilience as well. The minimal block size at which arithmetic codes perform better is around 400 characters, the overhead of Huffman codes over arithmetic coding is around 0.7% for large alphabets. Their conclusion was that the advantage of arithmetic codes over Huffman codes is so small that it may often be negligible.

For small, for instance binary, source alphabets Huffman codes perform poorly — they give 700% larger file sizes in comparison to arithmetic codes. But, in that case, run length encoding [64] often improves the compression ratio. After remodeling of the source, arithmetic codes are better than Huffman codes by only 0.5%. If probabilities of letters are inaccurate, Huffman codes perform better — they are less sensitive to such inaccuracies. Huffman codes are twice faster to compress and up to ten times faster to decompress. Adaptive Huffman codes are comparable to arithmetic coding in terms of compression speed, but they are four times faster to decompress. The code itself is easier to communicate in case of Huffman codes. Huffman codes are also more robust against errors. They resynchronize after an error and arithmetic codes do not (or not easily).

Even though the decoding of Huffman codes is fast, at least in comparison with arithmetic coding, there have been some research on ways to accelerate it. For instance Klein [35] introduced a data structure for fast decoding of messages encoded with canonical Huffman codes — codes in which longer codewords always precede lexicographically shorter ones. His representation reduces storage requirements for a code from $O(N)$ to $O(\log^2 N)$ if the longest codeword is of $O(\log N)$ length. The reduction of the number of bit operations speeds up the decoding by about 50%.

1.3 Synchronization of finite automata

Automata synchronization is a research area in *finite automata* field. In this work synchronization of Huffman codes is described as synchronization problem for a certain class of automata.

A finite automaton $\mathcal{A}$ is a triple $\langle Q, \Sigma, \delta \rangle$, where Q is the set of states of the automaton, Σ is an alphabet used for transitions and $\delta : Q \times \Sigma \to Q$ is a transition function. The automaton starts in a certain state $q \in Q$ and reads consecutive letters of an input word. The letters of the input word are elements of Σ. The automaton in state $q \in Q$ proceeds to state $\delta(q, a)$ upon reading letter a. After reading all the

letters of the input word w the automaton is in a certain state $q' \in Q$. We say that the word w *brings* the automaton $\mathcal{A}$ from the state q to q'.

A synchronizing string for a finite automaton $\langle Q, \Sigma, \delta \rangle$ is a string, s, that brings all states to one particular state, that is $\delta(q_1, s) = \delta(q_2, s)$ for any states $q_1, q_2 \in Q$. An automaton is called *synchronizing* if it has a synchronizing string. The famous Černý conjecture [14] states that a synchronizing finite automaton with N states has a synchronizing string of length $(N-1)^2$. Černý, however, proved in [14] only an exponential upper bound on the length of a synchronizing string.

Although there are proofs for certain classes of automata, for instance in [1, 32, 4], the problem remains open (in September 2008). There are some rougher bounds on the length of the shortest synchronizing string. For instance, Pin [56] proved that $\frac{1}{6}(N^3-N)$ is an upper bound. Some research has also been done to find automata with long shortest synchronizing strings. Černý [14] constructed a series of automata with the shortest synchronizing string of length $(N-1)^2$. Ananichev et al. [2] considered how long a synchronizing string can be if there is a letter that reduces the number of states by two. Trahtman [70] searched for worst-case automata by analyzing all possible automata of a given size.

Eppstein [18] presented an algorithm for testing in $O(N^2)$ whether an automaton with N states is synchronizing. He also presented an $O(N^3)$ algorithm for the construction of a synchronizing string of length $O(N^3)$ for a synchronizing automaton. The complexities given here are under condition that the alphabet is of constant size. Eppstein proved that the problem of testing whether an automaton has a synchronizing string of length less or equal m, for a given number m, is NP-complete.

An overview of the area of automata synchronization is given in [63, 45]. An in-depth presentation of the state of the art of the field can be found in [59].

1.4 New results

The dissertation contains a number of new results and novel ideas, the overview of which is presented in the next sections. Some of them were published in the following papers:

- M. T. Biskup, "Guaranteed synchronization of Huffman codes," in *Proc. 18th IEEE Data Compression Conference (DCC'08)*, pp. 462–471, IEEE Computer Society, (Los Alamos, CA, USA), 2008, [6].

- M. T. Biskup, "A word that does not appear in the encoded message as a resynchronization marker," in *Proceedings of the IEEE Information Theory Workshop*, Porto, Portugal, 2008, [8].

- M. T. Biskup, "Shortest Synchronizing Strings for Huffman Codes" in *Mathematical Foundations of Computer Science 2008,* E. Ochmański and J. Tyszkiewicz, eds., *Lecture Notes in Computer Science* **5162**, pp. 120–131, Springer, 2008, [7].

- M. T. Biskup, "Synchronization of Huffman Codes," 2008, unpublished.

1.4.1 Guaranteed synchronization of Huffman codes

Statistical synchronization of Huffman codes does not give any bound on the synchronization delay. Under certain conditions a decoder of even the best code remains unsynchronized for arbitrary number of bits. Therefore, no assumption can be made on the length of the propagation of a bit error, or on the number of incorrectly decoded bits for a decoder that starts at an arbitrary bit of the encoded message.

For Huffman codes a bound on the synchronization delay can be easily enforced with a synchronizing codeword (SC), by inserting it in regular intervals. This requires that that the synchronizing codeword does not correspond to a letter of the source alphabet and may be skipped by the decoder. Unfortunately, this assumption leads to suboptimal codes and the existence of a synchronizing codeword often requires a modification of the code and may introduce additional redundancy.

This dissertation introduces two variations of Huffman coding that guarantee synchronization of a decoder after processing at most L bits, L being a parameter (Sections 3.4 and 3.6). The redundancy introduced by the algorithms depends on the synchronization properties of the code and on the encoded data. It is equal zero if the encoded data always cause a decoder to resynchronize spontaneously in about L bits. The redundancy per encoded bit can also be made arbitrarily small by increasing the number L.

For the first method (Section 3.4) it is necessary that the decoder always knows the position of the bit being currently decoded. The method is simple and efficient. The overhead over normal Huffman coding in terms of processing time is only about 20%. The other method (Section 3.6) does not require any positional knowledge in the decoder, so it can also be used to limit propagation of bit-insertion or bit-deletion errors. It is done for the cost of increased processing time.

Both methods are new. The problem of ensuring limited synchronization delay for Huffman codes has been considered before [40, 28], but the codes used in other work were suboptimal. The idea of utilizing statistical synchronization of Huffman codes to reduce the inserted redundancy is also new.

The methods are asymptotically optimal in terms of computational complexity, because the number of additional operations is proportional to the length of the encoded message, independently of the code size. The methods are efficient and rather simple to implement, so they can be used in practice. The first method was applied to parallel Jpeg decompression.

1.4.2 Estimating the synchronization delay

The dissertation presents a method for a decoder that starts at an arbitrary bit of a Huffman encoded message to find out when it is synchronized (Section 3.2). This problem has not been considered before and the algorithm is also a novelty. The method is worst-case time optimal and the estimation of the resynchronization position cannot be improved without additional knowledge about data before the start position of the decoder.

1.4.3 Strong synchronization

Lam and Kulkarni [40] proposed insertions of an *extended synchronizing codeword* (ESC) in regular intervals to guarantee the limited synchronization delay. Not only does ESC allow a decoder to resynchronize, but also to realize that ESC has been received.

They gave an algorithm for modifying a Huffman code to introduce an ESC. Their modification, however, destroys the optimality of Huffman codes. The redundancy introduced by their method is nonzero even if no ESC is inserted, and is proportional to the length of the message.

In this dissertation, another way of choosing a bit string that has the same functionality as ESC is presented. This string, called a *resynchronization marker* (RM), can always be recognized by a decoder and then the decoder is able to recover synchronization. The error propagation may then be controlled by RM's insertions into the encoded message, either in regular intervals or using other schemes. With the method presented, the decoder is always aware of receiving the RM, unlike in case of synchronizing codewords, so it may do some error concealment procedure to correct the number of decoded symbols (see [40]). The method is general and works for any variable length code, even for codes that are not statistically synchronizable. If no markers are inserted, the redundancy grows only logarithmically with the length of the message (in case of ESC, the redundancy grows linearly).

Even though this method solves the same problem as the extended synchronizing codeword, introduced by Lam and Kulkarni [40], this solution is new.

1.4.4 Parallel Huffman decoding

In the era of multiprocessor personal computers it is important to decode compressed data in parallel. For that, compressed data has to be split into parts, each of them to be processed separately by a different CPU. In case of Huffman codes, decoding a fragment of compressed data requires that the processor is synchronized at start of the fragment.

Klein and Wiseman [38] developed a method for parallel decoding of Huffman codes that was based on statistical synchronization (see also Section 1.2.7). Even though the decoder may be unsynchronized at start of its fragment, after some bits it resynchronizes and then it decodes correct data. The prefix decoded incorrectly is later decoded by some other processor.

Even though the method works well in most cases, unfortunate conditions may cause a processor not to recover synchronization before the end of its fragment. In such case the work of this processor is wasted. It is therefore desirable to limit such synchronization delay.

The new method of parallel Huffman decoding, introduced in this dissertation, uses one of the techniques for guaranteed synchronization of decoders. For the price of a little redundancy in encoded data, it assures that decoders always resynchronize quickly, and, even in the worst case, only a little part of the work of each processor is dropped. The method is compared with simple schemes of dividing Huffman data

into blocks. Test show that the redundancy introduced by the new method is from ten to a hundred times better than in case of the other techniques, depending on the granularity of the division.

1.4.5 Bounds on the length of a synchronizing string

The equivalence of Huffman code synchronization and synchronization of certain finite automata, called Huffman automata, is shown in this dissertation. Further, synchronization properties of these automata are investigated, for instance an analysis of the length of the shortest synchronizing string is conducted.

The problem of bounding the length of the shortest synchronizing string for finite automata has been present for decades (see Section 1.3), in particular the famous Černý conjecture still remains an open problem. Certain classes of automata have been considered, and, for some of them, Černý conjecture has been proved. The class of automata analyzed in this dissertation, to the author's best knowledge, has not been studied before. An upper bound on the length of the shortest synchronizing string for this class of automata is presented. The bound is better than the best known $O(N^3)$ bound (N is the number of states of the automaton, or, equivalently, the number of codewords in the code). Even though Černý conjecture has not been proved for all automata of this class, in most cases this bound is better.

This dissertation includes also an upper bound on the length of the shortest *merging string* for a set of two states of a Huffman automaton, one of them being the root of the code's tree. A merging string for a set of states is a string that brings all states of the set to one particular state. The proof is constructive and provides an efficient algorithm for the construction of the shortest merging string for such nodes.

An efficient algorithm for answering whether a code is synchronizing is presented. Its complexity, $O(\sum_i |w_i|)$, where w_i are codewords, is better than the complexity of Eppstein's algorithm [18], $O(N^2)$. Also an algorithm for the construction of a synchronizing string for a Huffman automaton is introduced. Its complexity, roughly $O(\sum_i |w_i| \log^2 N)$, is better than the complexity of the corresponding Eppstein's algorithm [18], $O(N^3)$, the best known at the moment.

The dissertation contains results of numerical search for worst-case codes in terms of the length of the shortest synchronizing or merging string. Three interesting classes of Huffman codes were found. For them, the exact lengths of the shortest synchronizing and merging strings are given. These codes give a lower bound on the possible upper bounds of the length of the shortest synchronizing or merging string. It is conjectured (but, unfortunately, not proved) that these classes of codes are the worst-case codes for any code size. It is interesting that the length of the synchronizing and merging strings for the worst-case codes is much lower than the general bound proved, which shows that this upper bound can still be improved. This problem remains open.

1.4.6 Algorithms

There are a number novel and efficient algorithms presented in the dissertation:

1. two algorithms for finding all the synchronizing codewords for a Huffman code,
2. an algorithm for testing whether a code has a synchronizing string,
3. an algorithm for finding a short synchronizing string for a code,
4. an algorithm for estimating the maximum synchronization delay of any decoder in a Huffman-encoded message.

The problems 1 and 4, to the authors best knowledge, have not been considered before. For the problem 2, there is the algorithm of Capocelli et al. [12] and the algorithm of Eppstein [18]. The latter is more general than the one of Capocelli et al. and has better complexity. For the problem 3, another algorithm of Eppstein [18] can be used. The algorithms for problems 2 and 3 presented in this dissertation have better complexity than the algorithms of Eppstein.

1.5 Thesis organization

Chapter 2 contains the definitions used throughout the dissertation. It introduces the language used to describe results given in further chapters. Chapter 3 describes new methods to limit synchronization delay for decoders of Huffman-encoded data. It also contains results of tests of the methods and a description of their possible applications. Chapter 4 is dedicated to synchronizing strings and synchronizing codewords of Huffman codes and their relation with automata synchronization. Chapters 3 and 4 are, with few exceptions, independent of each other.

Chapter 2

Definitions and notation

2.1 Words

An *alphabet* is a nonempty set. The elements of an alphabet are called *letters*. Alphabets are denoted by capital Greek letters, for example Σ. The *size* of an alphabet Σ is the cardinality of the set Σ, denoted by $|\Sigma|$. In this work only finite alphabets are considered.

Example 2.1: The alphabet $\Sigma_b = \{0, 1\}$ is called the *binary alphabet*. It consists of two letters: 0 and 1. The size of Σ_b is 2: $|\Sigma_b| = 2$. The elements of Σ_b are called *bits*.

A *word* over an alphabet Σ is a finite sequence of letters of Σ. The empty word is denoted by ϵ. The set of all words over Σ is denoted by Σ^*. The set of all nonempty words over Σ is denoted by Σ^+. It follows that $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

A *concatenation*, v_1v_2, of two words v_1 and v_2 is a word that consists of the letters of v_1 followed by the letters of v_2. For instance, if $v_1 =$ '*abc*' and $v_2 =$ '*de*' then v_1v_2 = '*abcde*'. A concatenation of n words w is denoted by w^n, for example $1^3 = 111$.

If $w = v_1v_2$ then v_1 is called a *prefix* of w and v_2 is called a *suffix* of w. A prefix (suffix) v of w is called *proper* if $v \neq w$. $|w|$ denotes the number of letters in w — the *length* of w. If $w = v_1v_2v_3$ then v_2 is called a *subword* of w.

A word w is often written as $w = w_0w_1 \dots w_{k-1}$ — the concatenation of letters. Each letter has an associated index, counting from 0. Substrings of w will be referred to using the following definition:

Definition 2.2. *Position* i in a word w is the space between the letters $i - 1$ and i of w, counting from 0. Position 0 is before the 0-th letter.

The subword of a word w from position p to q is denoted by $w[p..q)$. The suffix of w that starts at position p is denoted by $w[p..)$. The prefix of w that ends at position p is denoted by $w[..p)$.

Example 2.3: Let w='*abc*' be a word over the alphabet $\Sigma = \{a, b, c\}$. The prefixes

of w are:

$$w[..0) = w[0..0) = \epsilon,$$
$$w[..1) = w[0..1) = \text{`}a\text{'},$$
$$w[..2) = w[0..2) = \text{`}ab\text{'},$$
$$w[..3) = w[0..3) = \text{`}abc\text{'}.$$

The marks " ' " and " ' " will be omitted in most cases. The suffixes of w are:

$$w[3..) = w[3..3) = \epsilon,$$
$$w[2..) = w[2..3) = c,$$
$$w[1..) = w[1..3) = bc,$$
$$w[0..) = w[0..3) = abc.$$

The only other subword of w is $w[1..2)$=‘b’.

2.2 Codes

Definition 2.4. Let Σ and Γ be two alphabets. A *code* is a set $C \subseteq \Gamma^+$ of *codewords* with a surjective mapping $c : \Sigma \to C$.

The set Σ is called the *source alphabet*, and Γ is called the *destination alphabet*. If $\Gamma = \{0, 1\}$, C is called a *binary code*.

Both c and C are used to refer to the code. In some cases the mapping c is unimportant and then the set C, alone, is called a code.

The size of the code, $|C|$, is denoted by N_C, or just N if the code C is known from the context.

Example 2.5: Let the source alphabet be $\Sigma = \{a, b, c\}$ and the mapping $c_3 : \Sigma \to \Sigma_b^+$ be defined by

$$c_3(a) = \text{`10'}, \qquad c_3(b) = \text{`01'}, \qquad c_3(c) = \text{'01'}. \tag{2.1}$$

This is a binary code. Note that the mapping c_3 is not injective.

The mapping c of a code can be extended to strings of letters as

$$c(a_0a_1 \dots a_k) = c(a_0)c(a_1) \dots c(a_{k-1}) \quad \text{for } a_i \in \Sigma. \tag{2.2}$$

Any word w over the source alphabet is called a *source message*. The word $c(w)$, which is a word over the destination alphabet, is called the message w encoded with the code c, or just the *encoded message* in case the code and the source message are known from the context. Usually the source message is denoted by $\mathcal{M}$ and the encoded message by $\mathcal{E}$.

Example 2.6: The string ‘100101’ is the message ‘abc’ encoded with the code c_3. But ‘100101’ is also the message ‘acb’ encoded with the code c_3.

Definition 2.7. A code c is *uniquely decodable* if the extended mapping (2.2), $c : \Sigma^* \to \Gamma^*$, is injective.

Unique decodability means that having a code and a message encoded with that code, one can always find the unique source message that produces the encoded message. Note that not all strings over the destination alphabet have to be messages encoded with a particular code, for instance, no message encoded with c_3 produces 111.

Example 2.8: The following code C_4 is uniquely decodable:

$$c_4(a) = \text{‘00’}, \quad c_4(b) = \text{‘01’}, \quad c_4(c) = \text{’1’}. \tag{2.3}$$

Indeed, if we have a binary string, we can reconstruct the source message by analyzing consecutive bits. If the string starts with 1, the first symbol must be c. Otherwise we look at the second symbol. If it is 0 then the we decode a, otherwise b. The remaining letters can be decoded in the same way.

A *variable-length code* is a code C that contains at least two codewords of different length, that is the set $\{|w| : w \in C\}$ has cardinality greater than 1. Otherwise a code is called a *fixed length code*. The code c_3 is a fixed length code while c_4 is a variable length code.

Note that we usually consider a class of codes that have the same properties or that are created with a common algorithm. Such a class may contain both variable length codes and fixed length codes. In this case, fixed length codes are special instances of these codes. The codes will therefore be called “variable length codes”, even though some of them may be fixed length. For instance, Huffman codes (see Section 2.5) may be variable length or fixed length, but they will be called variable length codes, because fixed length Huffman codes only exist under additional assumptions about the number of letters in a code and about the probability distribution on letters.

Definition 2.9. A *prefix code* is a code such that no codeword is a prefix of another codeword.

Prefix codes are always uniquely decodable [64]. The code c_4 is a prefix code and c_3 is not, because $c_3(b)$ is a prefix of $c_3(c)$. The property described by Definition 2.9 is called the *prefix property*. Prefix codes are also called *prefix-free*.

Definition 2.10. A *complete binary prefix code* is a binary prefix code such that if w is a proper prefix of some codeword then both $w0$ and $w1$ (concatenation) are prefixes of some codewords.

Example 2.11: Let $w =$ ‘10’ be a codeword of a complete binary prefix code. It has two proper prefixes: ϵ and 1. Then $w_1 = 0, w_2 = 1, w_3 = 10$ and $w_4 = 11$ are also prefixes of some codewords. This is obviously true for w_2 and w_3, as they are prefixes of w.

Figure 2.1: A complete binary tree

Example 2.12: The code c_4 is a complete binary prefix code. The code c_3 is not, because it is not a prefix code. The code:

$$c_5(a) = 00, \quad c_5(b) = 01, \quad c_5(c) = 11, \tag{2.4}$$

is not a complete binary prefix code, because it is not complete. Indeed, the word 1 is a prefix of the codeword 11, but the word 10 is not a prefix of any codeword.

Complete binary prefix codes will be referred to as *Huffman codes* in this dissertation. They can also be described in terms of complete binary trees, which are introduced in the next section.

Definition 2.13. A *Huffman code* is a complete binary prefix code.

2.3 Trees

A *complete binary tree* is a tree with each node being either an *internal node* with two children, or a *leaf* node with no children. Each left outgoing edge is labeled with 0 (0-edge) and each right outgoing edge is labeled with 1 (1-edge). The root of a tree is denoted by ε.

Definition 2.14. The *label* of a node n in a complete binary tree is the string $\pi(n)$ formed of edge labels on the path from the root to n.

Labels of nodes in a given tree are unique and they will be used to refer to the nodes. We have $\pi(\varepsilon) = \epsilon$, the label of the left child of the root is 0 and the label of the right child of the root is 1.

The height of a tree T is denoted by h_T. In most cases, the tree T will be known from the context and the subscript will be omitted.

Example 2.15: A complete binary tree is presented in Figure 2.1. The labels, $\pi(n)$, for leafs of the tree are (from left to right) 00, 01 and 1. These are exactly the elements of the Huffman code C_4 from Example 2.8. It will be shown later that there is one-to-one correspondence between complete binary trees and Huffman codes.

2.4 Automata

Definition 2.16. A *finite automaton* is a triple $\langle Q, \Sigma, \delta \rangle$, where Q is a finite set of *states*, Σ is an alphabet and $\delta : Q \times \Sigma \to Q$ is called the *transition function*.

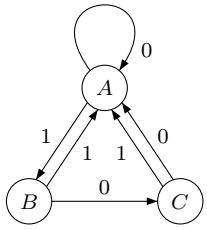


Figure 2.2: A finite automaton.

A finite automaton is always in some state $q \in Q$. It takes a word $w \in \Sigma^*$ as its input and makes transitions to other states, according to the transition function. The automaton in state p proceeds to state $\delta(p, a)$ upon reading letter a.

The above definition, contrary to common definition, omits the initial and final states because they are unimportant in this work.

Example 2.17: Figure 2.2 shows a finite automaton $\langle Q, \Sigma, \delta \rangle$ with $Q = \{A, B, C\}$, $\Sigma = \{0, 1\}$ and the transition function δ defined by:

$$\begin{aligned} &\delta(A,0) = A, && \delta(B,0) = C, && \delta(C,0) = A,\\ &\delta(A,1) = B, && \delta(B,1) = A, && \delta(C,1) = A. \end{aligned}$$

The transition function δ can be extended by induction to δ^*, that describes transitions for finite binary strings:

$$\delta^*(q, b_0 \dots b_{k-2} b_{k-1}) = \delta(\delta^*(q, b_0 \dots b_{k-2}), b_{k-1}), \tag{2.5}$$

$$\delta^*(q, \epsilon) = q. \tag{2.6}$$

Example 2.18: For the Example 2.17 automaton the following holds:

$$\begin{aligned} \delta^*(A, 010) &= C,\\ \delta^*(A, 101) &= A. \end{aligned}$$

We say that a word w *brings* a state n to a state n' if $n' = \delta^*(n, w)$. We also say that n' is the result of *applying* w to n.

For the set S of states of an automaton we denote,

$$\delta(S, a) = \{\delta(q, a) | q \in S\}. \tag{2.7}$$

The same convention is used for δ^*.

Definition 2.19. Let $\mathcal{A} = (Q, \Sigma, \delta)$ be a finite automaton. A *synchronizing string* for $\mathcal{A}$ is a word w such that $|\delta^*(Q, w)| = 1$.

Definition 2.20. An automaton is *synchronizing* if it has a synchronizing string.

Definition 2.21. Let $\mathcal{A} = (Q, \Sigma, \delta)$ be a finite automaton and let R be a set of states for $\mathcal{A}$. A *merging string* for R is a word w such that $|\delta^*(R, w)| = 1$.

The definition of a synchronizing and merging strings are closely related. A synchronizing string is a merging string for any nonempty set of states. A word w is a synchronizing string if and only if it is a merging string for the set of all states.

Example 2.22: The string 00 is a synchronizing string for the Example 2.17 automaton. The string 1 is a merging string for the states B and C of this automaton. The string 00 is also a merging string for these states.

The set of all subsets of a set S is denoted as $\mathcal{P}(S)$.

Definition 2.23. Let $\mathcal{A} = (Q, \Sigma, \delta)$ be a finite automaton. The *power automaton* for $\mathcal{A}$, denoted as $\mathscr{P}(\mathcal{A})$, is the automaton $(\mathcal{P}(Q), \Sigma, \delta_{\mathscr{P}})$ such that $\delta_{\mathscr{P}}(S, a) = \delta(S, a)$ for $S \in \mathcal{P}(Q)$.

The states of the power automaton $\mathscr{P}(\mathcal{A})$, that is sets of states of the automaton $\mathcal{A}$, are called *configurations*.

Operation of the power automaton $\mathscr{P}(\mathcal{A})$ can be seen as movements of coins that lie on some states of the automaton $\mathcal{A}$. If the power automaton is in a state $S \subseteq Q$ (in configuration S), the coins lie on the states $q \in S$. Then, if the power automaton makes a transition by a letter a, the coins move accordingly to the transition function δ for the automaton $\mathcal{A}$ — a coin on a state p moves onto the state $\delta(p, a)$. If more than one coin goes to the same state only one of them is kept.

It easy to see that after applying letter a to configuration S, the set of states with coins is exactly $\delta_{\mathscr{P}}(S, a)$. This analogy helps to visualize the operation of a power automaton and gives some intuition. For instance, the string w is synchronizing if and only if applying w to the automaton $\mathcal{A}$ with a coin on each state results in just one coin remaining.

2.5 Huffman codes

Lemma 2.24 (folklore)**.** *Let T be a complete binary tree. Let*

$$\mathcal{C}(T) = \{\pi(n) | n \text{ is a leaf of the tree } T\}. \tag{2.8}$$

The set $\mathcal{C}(T)$ is a Huffman code.

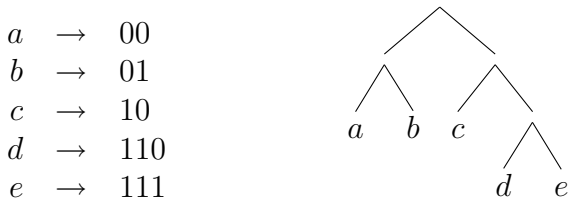
The codewords of the code $\mathcal{C}(T)$ are leaf labels for the tree T. Also, for each Huffman code C there is a complete binary tree T such that $C = \mathcal{C}(T)$.

Definition 2.25. The *Huffman tree* T_C for a Huffman code C is a binary tree such that

$$C = \mathcal{C}(T_C) = \{\pi(n) | n \text{ is a leaf of the tree } T_C\}. \tag{2.9}$$

The subscript C in T_C will usually be omitted.

Let c^{-1} be the inverse for the code mapping c. It maps codewords of C to letters (see Definition 2.4). We can associate the source letter $c^{-1}(\pi(n))$ with each leaf n of a Huffman tree T_C. With these characters the Huffman tree contains all the information about the Huffman code.



(a) Huffman code C_1 (b) Huffman tree for the code C_1

Figure 2.3: Huffman code C_1 from Example 1.1 and its Huffman tree.

Example 2.26: Huffman code C_1 from Example 1.1 and its Huffman tree are shown in Figure 2.3. The code transforms the 5-element source alphabet $\{a, b, c, d, e\}$ into binary strings. The labels on the edges of the tree are omitted.

The encoding with a Huffman code C for a source message $\mathcal{M}$ can be computed by substituting each $\mathcal{M}$'s letter x by its codeword $c(x)$.

Example 2.27: Let us consider the Huffman code C_1, presented in Figure 2.3, and let the source message be

$$\mathcal{M} = \text{`}bbeaaebcec\text{'}. \tag{2.10}$$

The result of encoding this message with the code C_1 is:

$$\mathcal{E} = c_1(\mathcal{M}) = \underbrace{0\,1}_{b}\,\underbrace{0\,1}_{b}\,\underbrace{1\,1\,1}_{e}\,\underbrace{0\,0}_{a}\,\underbrace{0\,0}_{a}\,\underbrace{1\,1\,1}_{e}\,\underbrace{0\,1}_{b}\,\underbrace{1\,0}_{c}\,\underbrace{1\,1\,1}_{e}\,\underbrace{1\,0}_{c}. \tag{2.11}$$

The bits that form codewords are grouped together.

A sequence encoded with a Huffman code can be decoded using the Huffman tree for the code. We start in the root and read consecutive bits. If the bit is 0, we descend to the left child. If the bit is one, we descend to the right child. When we reach a leaf we output the letter from the leaf and restart from the root.

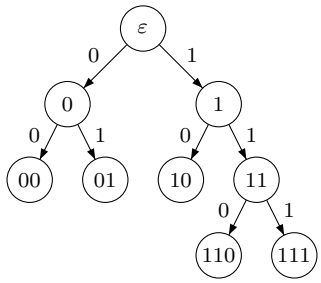
This procedure can be seen as transitions of a finite automaton that, in addition, outputs some data. We can formalize the decoding process by introducing *Huffman automata*.

Definition 2.28. A *Huffman tree automaton* for a Huffman code C is a finite automaton formed of all the nodes in the Huffman tree for the code C. The transition function $\delta_T(n, b)$, $b \in \{0, 1\}$, brings the automaton from the node n to the b-edge child of n. The automaton finishes its operation upon reaching a leaf.

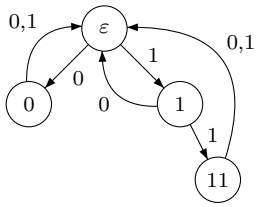
Note that the function δ_T is not a total function, because it is not defined for leaves. This is irrelevant to the operation of the Huffman tree automaton as it finishes upon reaching a leaf.

The above description of decoding can be expressed in terms of the Huffman tree automaton. The automaton starts in ε and reads consecutive bits of the input, until it reaches a leaf. Then it outputs the letter from the leaf and restarts in ε.

If the decoded letters are irrelevant and only the state of the decoder is important, the following definition may be used:



(a) Huffman tree automaton for the code C_1



(b) Huffman automaton for C_1

Figure 2.4: Huffman automaton for the code C_1 from Figure 2.3.

Definition 2.29. A *Huffman automaton* $\mathcal{T}_C$ for a Huffman code C is the Huffman tree automaton for C with all the leaves merged with the root.

This means that whenever there is a transition from a node n to a leaf in the Huffman tree automaton, there is a transition from n to ε in the Huffman automaton. The leaves are not present in the Huffman automaton.

It is important to note that the values of δ_T are all the nodes of the tree but ε. On the other hand, the transition function $\delta_\mathcal{T}$ of the Huffman automaton gives only internal nodes of the tree as values.

The subscript C in $\mathcal{T}$ will usually be omitted. Also $\delta_\mathcal{T}$ will usually be denoted simply by δ.

We say that w *brings* a node n to the root if $\delta^*(n, w) = \varepsilon$. We also say that w brings a node n to a leaf if $\delta^*(n, w) = \varepsilon$ and w is not empty. This is so because leaves are equivalent to the root in the Huffman automaton. We say that w brings a node n to n' without *loops* (we also say "without passing through a leaf") if none of the nodes

$$\delta^*(n, w[..1)), \delta^*(n, w[..2)), \ldots, \delta^*(n, w[..|w| - 2)) \tag{2.12}$$

is the root.

Example 2.30: Huffman tree automaton for the code C_1 is presented in Figure 2.4(a). It has the same shape as the Figure 2.3(b) tree. The corresponding Huffman automaton is presented in Figure 2.4(b).

To decode the sequence $c_1(\mathcal{M})$ from (2.11), the tree automaton starts in the root (the node ε). The first bit of $c_1(\mathcal{M})$ is 0, so the automaton moves to the node 0, accordingly to the transition function. The next bit, 1, forces a move to the leaf node 01, that corresponds to letter b (see Figure 2.3(b)), thus b is decoded and the automaton starts again from the root. The next two bits, 01, decode b again. Next, 111 makes the automaton reach the leaf 111 and decode letter e. As we can see, whenever the automaton starts processing a new codeword it is in the state ε.

2.6 Synchronization modeling

Bit errors may influence the operation of a decoder and may put it into an incorrect state. This results in error propagation.

Example 2.31: Let us see what happens if decoding starts from the second bit of $c_1(\mathcal{M})$ from Example 2.27. The second and third bit of $c_1(\mathcal{M})$, '10', are decoded as the letter c, then the bits 111 as e, 10 as c, etc. This results in the following decoding (the superscripts denote the position number):

$$c_1(\mathcal{M}) = {}^{0}0\underbrace{1\,0}_{c}\underbrace{1\,1\,{}^{5}1}_{e}\underbrace{1\,0}_{c}\underbrace{0\,0}_{a}{}^{10}\underbrace{0\,1}_{b}\underbrace{1\,1\,0}_{d}{}^{15}\underbrace{1\,1\,0}_{d}{}^{18}\underbrace{1\,1\,{}^{20}1}_{e}\underbrace{1\,0}_{c}. \qquad (2.13)$$

The decoded string is *'cecabddec'* (compare with the source message (2.10): *'bbeaaebcec'*). As we can see, the decoder works incorrectly until the last two codewords. From position 1 to 18 the state of the decoder is different than the state of the decoder that would have started at the first bit. For instance, at position 3, the state of the above decoder is ε, while the state of the latter decoder would be 0. This incorrect state causes the incorrect decoding of the source message.

At position 18, the decoder is in the state ε. This is the same state as the state of the decoder that starts form the first bit of the message (we noted in Example 2.30 that a decoder is in ε each time it starts processing a new codeword). As the decoders are in the same state, the remaining bits will be decoded in the same way.

Example 2.31 shows that if a decoder starts inside a codeword, not only does it decode this particular codeword incorrectly, but also a number of following codewords. It may, however, happen that a decoder resynchronizes at some point. Then the error propagation is stopped and, from then on, the decoded symbols are correct. Such synchronization is a general property of most Huffman codes and will be discussed in depth.

We can formalize what was said in Example 2.31 with the following definitions.

Definition 2.32. A *decoder*, D_p, for a code C and an encoded message $\mathcal{E}$, starting at position p, is the Huffman automaton of the code C that uses the word $\mathcal{E}[p..)$ as its input.

We assume that the encoded message $\mathcal{E}$ and the code C are always known from the context, thus they are omitted in the notation for D_p.

Definition 2.33. The *correct decoder* is the decoder starting at position 0 in $\mathcal{E}$, i.e. decoder D_0.

The correct decoder decodes the original source message. As seen in Example 2.31, decoders that start in any other position may fail to decode the message correctly.

Definition 2.34. Decoder D_p at position p' is *synchronized* if its state at p' is the same as the state of the correct decoder at p'. Otherwise, decoder D_p is *unsynchronized*.

Formally, decoder D_p is synchronized if:

$$\delta^*(\varepsilon, \mathcal{E}[p, p')) = \delta^*(\varepsilon, \mathcal{E}[..p')). \tag{2.14}$$

A decoder that is synchronized at p' is synchronized at all $p'' \geq p'$. A synchronized decoder will continue decoding the same sequence of letters as the correct decoder.

Definition 2.35. Decoder D_p *resynchronizes* at position p' if it is unsynchronized at any position q, $p \leq q < p'$, and at position p' it is synchronized.

Note that when the decoder resynchronizes it is always in the state ε.

Definition 2.36. The *synchronization delay* for a decoder D_p is the number of bits read by D_p until it resynchronizes.

If D_p resynchronizes at p' then its synchronization delay is $p' - p$.

Example 2.37: The decoder in Example 2.31 is a D_1 decoder. The codewords decoded by this decoder are shown in (2.13). The codewords seen by the correct decoder, D_0, are shown in (2.11). It is easy to verify that the state of the decoder D_1 is different from the state of the decoder D_0 until position 18. The synchronization delay of D_1 is $18-1 = 17$. Decoder D_1 is unsynchronized at positions 1 to 17, resynchronizes at position 18 and is synchronized at positions 18 to 23, inclusive.

Note that in Chapter 1 the synchronization delay was expressed in terms of the number of symbols lost before resynchronization. From now on, we focus on the number of bits in the encoded message.

Now, let us consider the influence of a bit-flip error on the decoded string.

Example 2.38: Assume that the third bit of the message $c_1(\mathcal{M})$ from Example 2.27 was flipped giving message $c_1(\mathcal{M})_e$. The string $c_1(\mathcal{M})_e$ and its division into codewords is the following (the flipped bit is marked in bold):

$$c_1(\mathcal{M})_e = {}^{0}\underbrace{0\ 1}_{b}\ \underbrace{\mathbf{1}\ 1\ 1}_{e}\ {}^{5}\underbrace{1\ 1\ 0}_{d}\ \underbrace{0\ 0}_{a}\ {}^{10}\underbrace{0\ 1}_{b}\ \underbrace{1\ 1\ 0}_{d}\ {}^{15}\underbrace{1\ 1\ 0}_{d}\ {}^{18}\underbrace{1\ 1\ {}^{20}1}_{e}\ \underbrace{1\ 0}_{c} \tag{2.15}$$

The decoded message is *bedabddec* (compare with the source message (2.10): *'bbeaaebcec'*). The first letter, b, was decoded correctly. The error occurred in the second codeword and instead of b, the second decoded letter was e. The next letter is also wrong and the error propagates again until position 18. The decoder at positions from 3 to 17 is unsynchronized with respect to the correct decoder of the original message $c_1(\mathcal{M})$.

Note that there are two different messages, both processed by a D_0 decoder. We can see that even a single bit error propagates, because it causes the decoder to lose synchronization.

Synchronization recovery after a bit error proceeds in the same way as synchronization recovery for a decoder that started at some non-zero position in the original message. In Example 2.38, when the decoder of $c_1(\mathcal{M})_e$ is at position 6, it is in the

state ε and its further operation is equivalent to the decoder D_6 of $c_1(\mathcal{M})_e$. Since the suffix $c_1(\mathcal{M})_e[6..)$ is the same as $c_1(\mathcal{M})[6..)$, the resynchronization of $c_1(\mathcal{M})_e$-decoder D_0 is equivalent to resynchronization of $c_1(\mathcal{M})$-decoder D_6. In general, a decoder for an erroneous message is equivalent to D_x on the error-free message, for x being the first bit after the first incorrectly decoded codeword.

In the presence of errors the term *synchronization delay* means the number of erroneously decoded bits (sometimes erroneous characters). These are the bits starting at the beginning of the codeword where the error appeared until resynchronization.

2.7 Synchronizing strings and synchronizing codewords

Some Huffman codes have, so called, *synchronizing strings*. Such strings, when encountered by a decoder, always put it into synchronization.

Definition 2.39 ([65])**.** A string w_s is a synchronizing string for a Huffman code C if and only if ww_s is a sequence of codewords of C for any binary word w.

Definition 2.40. A *synchronizing Huffman code* is a Huffman code for which a synchronizing string exists.

An equivalent description of a synchronizing string is the following:

Lemma 2.41. *A string w_s is a synchronizing string for a Huffman code C if and only if:*

$$\forall_{n\in Q} \quad \delta^*(n, w_s) = \varepsilon, \tag{2.16}$$

where Q is the set of states of the Huffman automaton $\mathcal{T}_C$ and δ is the transition function for $\mathcal{T}_C$.

A string w_s satisfying the condition from Lemma 2.41 indeed puts any decoder into synchronization. By (2.16), a decoder always ends in the state ε, no matter what its state was before processing w_s. Since the correct decoder also finishes in the state ε, the state of the two decoders is the same, so the decoder is synchronized. The details of the proof are left to the reader.

Example 2.42: The string $w_s = 0110$ is a synchronizing string for the code C_1 from Example 1.1. In Examples 2.31 and 2.38 the string 0110 appeared just before the last two codewords (from position 14 to 18) and, indeed, it synchronized the decoders in each case.

To prove 0110 is synchronizing we have to consider the transitions of the Huffman automaton from any possible state by the word w_s. The resulting

state should always be ε. By inspection:

$$\varepsilon \xrightarrow{01} \varepsilon \xrightarrow{10} \varepsilon \tag{2.17}$$

$$0 \xrightarrow{0} \varepsilon \xrightarrow{110} \varepsilon \tag{2.18}$$

$$1 \xrightarrow{0} \varepsilon \xrightarrow{110} \varepsilon \tag{2.19}$$

$$11 \xrightarrow{0} \varepsilon \xrightarrow{110} \varepsilon \tag{2.20}$$

$$\tag{2.21}$$

This proves that 0110 is a synchronizing string.

By taking $w = \epsilon$ in Definition 2.39, we can see that a synchronizing string is a sequence of codewords.

Existence of a synchronizing string is a desired property of codes. Each time a synchronizing string appears in the message, all the previous synchronization errors are corrected.

Example 2.43: Let us consider sending a telegram — a short text message encoded with a Huffman code. Assume that the encoding for the word 'STOP' is a synchronizing string. Each appearance of this word stops propagation of previous errors. Therefore, if the word 'STOP' is sent instead of a period at the end of each sentence, any error that occurs within a sentence does not influence any other sentences.

Definition 2.44. A *synchronizing codeword* is a synchronizing string that is a codeword.

Example 2.45: The code C_1 from Example 1.1, does not have a synchronizing codeword. The codewords 00 and 10 are not synchronizing because 00 and 10 bring the state 0 to 0, rather than ε. 111 and 01 are not synchronizing, because they bring 1 to 1 (and not to ε). Finally 110 is not synchronizing, because it brings 1 to 0.

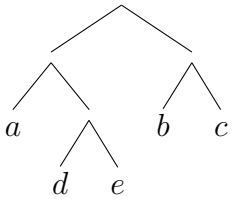
Example 2.46: The code C_2, Figure 2.5, has two synchronizing codewords: 010 and 011. The target state for the transition of the Huffman automaton for C_2 from any state by either 010 and 011 is ε.

Example 2.47: Synchronizing codewords improve the synchronization properties of a code. Assume that in Example 2.43 the code has a synchronizing codeword, and that it is the encoding of the '.' (period) character. Then, instead of the four-letter word *STOP* one can use just one character '.' to limit error propagation.

Example 2.48: The code C_6, Figure 2.6, does not have a synchronizing string at all. The codewords of C_6 are of length 2 or 4, so if the decoder is in a state 1 or 3 edges away from the root, any codeword brings it to another state 1 or 3 edges away from the root. As w_s is a string of codewords, it cannot bring such a node to the root.

a	$\rightarrow$	00
b	$\rightarrow$	10
c	$\rightarrow$	11
d	$\rightarrow$	010
e	$\rightarrow$	011

(a) Huffman code C_2

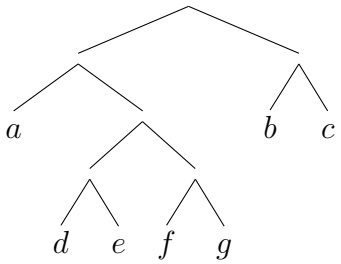


(b) Huffman tree for C_2

Figure 2.5: Huffman code C_2 from Example 1.2 and its and Huffman tree.

a	$\rightarrow$	00
b	$\rightarrow$	10
c	$\rightarrow$	11
d	$\rightarrow$	0100
e	$\rightarrow$	0101
f	$\rightarrow$	0110
g	$\rightarrow$	0111

(a) Huffman code C_6



(b) Huffman tree for C_6

Figure 2.6: Huffman code C_6 and its Huffman tree.

As seen in Example 2.48, there are Huffman codes that are not synchronizing. A subclass of these codes is described by the following lemma:

Lemma 2.49 (folklore, e.g. [22])**.** *If the greatest common divisor of the codewords' lengths is greater than 1, the code does not have a synchronizing string.*

The reverse is not true. There are codes without a synchronizing string for which the greatest common divisor of codewords' lengths is 1. Nevertheless, Schützenberger [65] proved that in this case there always exists a code with the same codewords' lengths that has a synchronizing string.

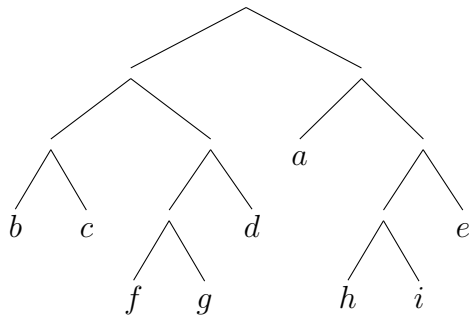
Definition 2.50 ([5])**.** A *biprefix* code is a prefix code that also meets the *suffix condition*: no codeword is a suffix of another codeword.

Example 2.51: The code C_7, Figure 2.7, is a biprefix code. By inspection:

- the codewords for letters f, g, h and i cannot be suffixes of other codewords because there are no codewords of length greater than 4;
- no codeword other than 10 ends with 10, so the codeword for a is not a suffix of another codeword;
- no codeword of length 4 ends with 11, so the codewords of d and e are not suffixes;
- finally, the codewords for b and c are not suffixes of the codewords f, g, h and i.

a	$\rightarrow$	10
b	$\rightarrow$	000
c	$\rightarrow$	001
d	$\rightarrow$	011
e	$\rightarrow$	111
f	$\rightarrow$	0100
g	$\rightarrow$	0101
h	$\rightarrow$	1100
i	$\rightarrow$	1101

(a) Huffman code C_7



(b) Huffman tree for C_7

Figure 2.7: Biprefix Huffman code C_7 and its Huffman tree.

It can be verified that the code C_7 does not have a synchronizing string, even though it has codewords of lengths 2, 3 and 4 and $GCD(2, 3, 4) = 1$.

Theorem 2.52 (folklore)**.** *No complete binary biprefix code of size greater than 2 has a synchronizing string.*

Proof. Let us assume the contrary, that w_s is a synchronizing string for such a code, that is any state is brought to ε by w_s. Let us consider a decoder in a state q_0 other than ε (there is such a state, because the code has at least three codewords, so there must be an at least 2-long codeword). Let $w_s = w_1 w_2 \dots w_k$, where w_i are codewords. Let q_1 be target state for q_0 after applying w_1, q_2 — after $w_1 w_2$, etc. It must be $q_l = \varepsilon$ and $q_{l-1} \neq \varepsilon$ for some $l \leq k$. Then w_l brings the node q_{l-1} to ε and the string $\pi(q_{l-1})w_l$ is a sequence of codewords. Let w be the last codeword in this sequence. Then either w is a suffix of w_l or w_l is a suffix of w, or w is equal w_l. None of these is possible. □

Synchronizing strings for Huffman codes are closely related to synchronizing strings of automata. The following theorem follows Lemma 2.41.

Theorem 2.53. *A synchronizing string for a Huffman code C is a synchronizing string for the Huffman automaton $\mathcal{T}_C$. A synchronizing string w_s for the Huffman automaton $\mathcal{T}_C$, such that w_s brings all nodes to the root, is a synchronizing string for the Huffman code.*

Corollary 2.54. *A Huffman code is synchronizing if and only if its Huffman automaton is synchronizing.*

2.8 Notation

The logarithms in this work are of base 2 unless explicitly stated otherwise.

The values T, $\mathcal{T}$, δ, δ^*, δ_T, N, h, ε, π depend on the code C. We assume that it is always clear from the context which code (or, equivalently, which Huffman tree) is being considered.

Chapter 3

Guaranteed Sychronization of Huffman Codes

3.1 Overview

In Huffman-encoded data a bit error propagates because the decoder loses synchronization with the coder. For many codes the decoder resynchronizes quickly in most cases. If a code has a synchronizing string or a synchronizing codeword, the decoder is always resynchronized after having processed such a string. Unfortunately, the existence of a synchronizing string or a synchronizing codeword does not guarantee that error propagation is limited to some fixed number of bits.

Example 3.1: In Example 2.47 the period corresponds to a synchronizing codeword. The decoder always resynchronizes after reading a period so the error propagation is always limited to one sentence. But without any a priori knowledge of the encoded message, one cannot assume anything about the length of sentences. Even if most of sentences in the message are short, there may be a long sentence, for instance occupying half of the message.

The situation is even worse in Example 2.43, where the encoding of the word *STOP* was a synchronizing string. It cannot be assumed that this word appears in the encoded message at all. To guarantee a limit on error propagation one can explicitly insert the synchronizing strings, the encodings of '.' or 'STOP', into the message. This, however, requires special handling, because the inserted strings have to be removed by the decoder. These strings also increase the length of the encoded message.

In Example 3.1 a simplification is made, because reading a synchronizing string is not a necessary condition for synchronization of a decoder. Nevertheless, sequences of bits with infinite synchronization delay are easy to construct.

Example 3.2: The following example is due to Takishima et al. [68]. Let us consider the probability distribution on letters a, b, c, d and e equal 0.3, 0.3, 0.2, 0.1 and 0.1, respectively. It is easy to see that the code C_1, Figure 2.3, is an optimal code for this probability distribution. In the message $\mathcal{M} = baecacbdba$ the frequencies of letters follow exactly the probability distribution. The encoding

of this message is the following:

$$c_1(\mathcal{M}) = \underbrace{0}_{b}\underbrace{1}_{a}\underbrace{0}_{e}\underbrace{0}_{c}\underbrace{1}_{a}\underbrace{1}_{c}\underbrace{1}_{b}\underbrace{1}_{d}\underbrace{0}_{b}\underbrace{0}_{a}\underbrace{1}\underbrace{0}\underbrace{0}\underbrace{1}\underbrace{1}\underbrace{1}\underbrace{0}\underbrace{0}\underbrace{1}\underbrace{0}\underbrace{0} \tag{3.1}$$

If the message is decoded from the second bit, the result is:

$$c_1(\mathcal{M}) = 0\underbrace{1}_{c}\underbrace{0}_{b}\underbrace{0}_{e}\underbrace{1}_{a}\underbrace{1}_{b}\underbrace{1}_{a}\underbrace{1}_{e}\underbrace{0}_{a}\underbrace{0}_{b}\underbrace{1}\underbrace{0}\underbrace{0}\underbrace{1}\underbrace{1}\underbrace{1}\underbrace{0}\underbrace{0}\underbrace{1}\underbrace{0}0 \tag{3.2}$$

If $\mathcal{M}$ is repeated infinite number of times, the decoder that starts from the second bit of the infinite message will never resynchronize and the decoded sequence will be totally wrong. For instance, the letter d does not appear in the incorrectly decoded message at all.

For a decoder that starts in the middle of a message it is important to know whether or not it has already resynchronized, to know if the decoded symbols are correct. In this chapter a method to achieve this goal is described (Section 3.2).

Then, we study synchronization properties of Huffman codes (Section 3.3), for instance the existence of a synchronizing string, and of messages encoded with Huffman codes (Section 3.5), for instance the maximum synchronization delay of some decoder. We present two methods for setting an upper bound on the synchronization delay of any decoder of a Huffman-encoded message, for the price of a small redundancy and increased processing time (Sections 3.4 and 3.6). The methods exploit the inherent tendency of Huffman codes to resynchronize and the additional bits are inserted only if the synchronization delay of some decoder would exceed the bound. In particular, if all decoders resynchronize quickly enough, the redundancy is zero.

Neither statistical synchronization of Huffman codes nor the two methods for guaranteed synchronization can provide the decoder with strong synchronization. A novel method for this problem is shown in Section 3.7.

All the methods described here are tested numerically and the results are presented in Section 3.8. Applications for methods from this chapter are discussed in Section 3.9, in particular limiting error propagation and parallel Huffman decoding. The latter is tested on Jpeg files (Section 3.10). The chapter is concluded in Section 3.11.

3.2 Estimating the synchronization delay

A decoder that starts at position p inside an encoded message $\mathcal{E}$ may be synchronized or unsynchronized at start. To see if the decoder is synchronized one has to decode the message from the beginning and check if the start position, p, is at a codeword boundary. In many cases the cost of decoding the whole prefix of the message is too high. It may also happen that the prefix is missing. It will be shown that in most cases the decoder can recover the correct codeword boundaries after having read only a small part of the message, starting at position p.

As mentioned before, the synchronization state of a decoder D_p depends on whether position p is a codeword boundary. This information is unknown at start.

Algorithm 3.1: Estimation of the synchronization delay.

Input: $\mathcal{E}$ — encoded message, p — decoder's start position.
Output: A position where decoder D_p is synchronized.

```
1 S ← ∅, i ← p;
2 while i < p + h do
3 |   S ← δ(S ∪ {ε}, E[i]);
4 |   i ← i + 1;
5 while |S| > 1 and i < |E| do
6 |   S ← δ(S, E[i]);
7 |   i ← i + 1;
8 return i
```

Nevertheless, the largest codeword's length is h, so at least one of the decoders $D_p, D_{p+1}, \ldots, D_{p+h-1}$ starts at a codeword boundary and is synchronized at start. It is enough to analyze the states of these h decoders at consecutive position in $\mathcal{E}$. When, at some position, all of them are found in the same state, they must all be synchronized.

The method is formalized in Algorithm 3.1. In lines 2 to 4, a set S with the current states of the h decoders is created while reading the first h bits. Then, in lines 5 to 7, transitions by consecutive bits are made until S reduces to just one state. At that moment, decoder D_p is synchronized and the algorithm returns the current position. Note that the algorithm accesses only the bits of $\mathcal{E}$ after position p.

Decoder D_p resynchronizes no later then at the position returned by the algorithm. It means that the algorithm returns an upper estimate of the synchronization delay for D_p. In some cases the algorithm may not detect the resynchronization until the end of the message. Tests show (Section 3.8), however, that such cases are rare and typically the algorithm stops after just a few tens of bits.

The worst-case time complexity for Algorithm 3.1 is $O(hd)$, where d is the length of the estimation of the synchronization delay found ($d = i - p$, with the notation of Algorithm 3.1).

The state transitions for decoders in the set S can, however, be computed more efficiently. The simplest optimization is to compute transitions by entire codewords, that is to use δ^* for codewords instead of δ for individual bits. With such optimization the computational complexity for Algorithm 3.1 is reduced to $O(hc)$ where c is the number of analyzed letters (decoded using the given code) before the algorithm finishes. The next section describes a method for precomputing in $O(N^2)$ the function δ^* for all codewords.

The worst-case time complexity for Algorithm 3.1 can still be improved. In Section 3.5.2 it is shown how the transition function for all the decoders from the set S by a single bit can be computed in constant time. It reduces the time complexity for this algorithm to $O(d)$. With such complexity, the algorithm is asymptotically optimal. Moreover, the returned position is also optimal if there is no a priori knowledge about the missing prefix of the encoded message.

3.3 The synchronization graph

In Algorithm 3.1 the transitions of the Huffman automaton were computed using a single bit at a time. To speed up computations, it is useful to know the transitions the Huffman automaton makes on reading whole codewords. We are interested in the values of $\delta^*(q, w_i)$ for each state q of the Huffman automaton and for each codeword w_i. This information will also be used to derive some properties of the code.

Definition 3.3. A *synchronization graph* for a Huffman code C is a directed graph, G_C, whose vertices are internal nodes of the Huffman tree T_C and edges are labeled with codewords (or, equivalently, with letters of the source alphabet). There is an edge $p \xrightarrow{w_i} q$ in G_C if and only if $\delta^*(p, w_i) = q$.

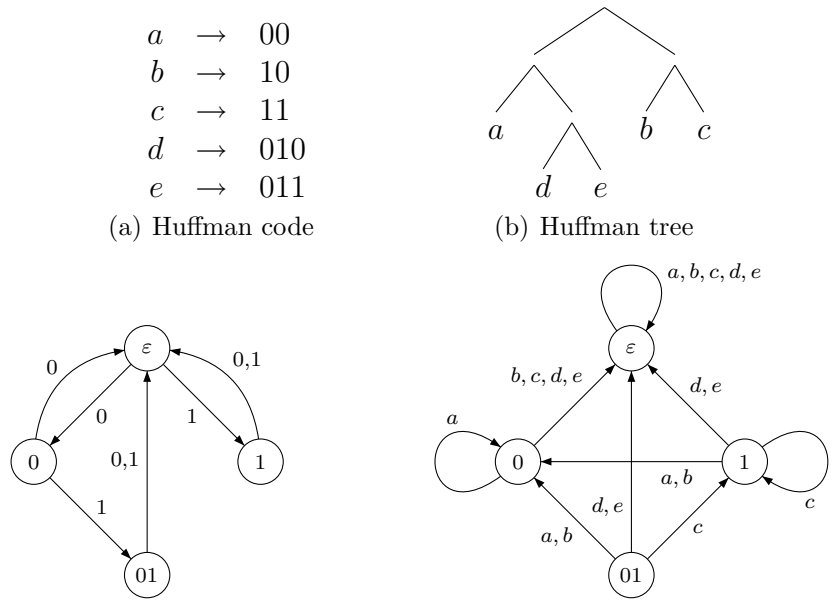
Note that instead of the codewords w_i it is preferable to use the corresponding letters $c^{-1}(w_i)$ to label edges of G_C. With such a representation the amount of data associated with a single edge is constant. For simplification, the codewords w_i are sometimes marked on edges, but the reader should be aware that, in fact, it is enough to store just the letters $c^{-1}(w_i)$. It is easy to see that there are exactly $N-1$ vertices in the graph, and exactly N edges go out of each vertex. The total size of the synchronization graph is $O(N^2)$.

Example 3.4: Huffman code C_2 and its synchronization graph are shown in Figure 3.1. Figure 3.1(a) presents the code, its Huffman tree is shown in Figure 3.1(b), the Huffman automaton is shown in Figure 3.1(c) and the synchronization graph is shown in Figure 3.1(d).

The most straightforward approach to compute the synchronization graph is to traverse the Huffman automaton from each state using each codeword. This takes time proportional to $O(N \sum |w_i|)$, where w_i are codewords. Although $\sum |w_i|$ may be of the order $N \log N$, in the worst case it is $\Theta(N^2)$, so the total processing time is $O(N^3)$.

It is clear that this approach is not optimal. Let two codewords, w_1 and w_2, share a common prefix v: $w_1 = vv_1$ and $w_2 = vv_2$. Then, for a node n, $\delta^*(n, w_1) = \delta^*(\delta^*(n, v), v_1)$ and $\delta^*(n, w_2) = \delta^*(\delta^*(n, v), v_2)$. It is enough to compute $\delta^*(n, v)$ once and use this value to compute $\delta(n, w_1)$ and $\delta(n, w_2)$. There is a place for improvement because the codewords form a complete binary tree, so they share many common prefixes.

This optimization is used in Algorithm 3.2. For each node v of a Huffman tree T procedure SearchTransitions is called from the main loop in line 11. It computes the targets for edges going out of vertex v using the optimization described. The last argument of SearchTransitions is the vertex for which the transitions are being computed. The first one is the current node in the Huffman tree traversal and it is set to v at start. The second argument, y, determines the common prefix, $\pi(y)$, of codewords that has just been used in the traversal of the tree. Procedure SearchTransitions recursively extends this prefix by 0 and by 1 in lines 8 and 9. The fields left and right for a node x give the 0-child and the 1-child of x, respectively.



(a) Huffman code (b) Huffman tree

(c) Huffman automaton (d) Huffman synchronization graph

Figure 3.1: Huffman code C_2, its Huffman tree, Huffman automaton and synchronization graph.

Algorithm 3.2: Construction of the synchronization graph.

Input: T — Huffman tree.
Output: Transitions $\delta^*(q, w)$ for any state q and any codeword w.

```
1  procedure SearchTransitions(x, y, x_0);
2    if x is a leaf then
3      SearchTransitions(ε, y, x_0);
4    else
5      if y is a leaf then
6        add edge [x_0 --π(y)--> x] to the graph (i.e. set: δ*(x_0, π(y)) ← x);
7      else
8        SearchTransitions(x.left, y.left, x_0);
9        SearchTransitions(x.right, y.right, x_0);
10 end procedure;
11 forall internal node v ∈ T do
12   SearchTransitions(v, ε, v);
```

The cost of Algorithm 3.2 is $O(N^2)$ because procedure SearchTransitions is called $N-1$ times from the algorithm's body and each such call is a DFS traversal of the Huffman tree given in the second parameter and thus takes $O(N)$ operations.

The synchronization graph can be used to find various properties of the code, such as the existence of a synchronizing string or the set of all synchronizing codewords.

Theorem 3.5. *A word w is a synchronizing codeword for a Huffman code if for any node n of the synchronization graph there is an edge $n \xrightarrow{w} \varepsilon$.*

Example 3.6: In the synchronization graph for the code C_2, Figure 3.1(d), all the edges labeled with d and e end in ε. Their corresponding codewords, 010 and 011, are synchronizing. Edges labeled with other letters do not always point to ε, so their codewords are not synchronizing.

With a synchronization graph we are able to find all the synchronizing codewords in $O(N^2)$ operations. More efficient methods for finding all synchronizing codewords without building the synchronization graph explicitly are shown in Chapter 4.

Theorem 3.7. *A Huffman code has a synchronizing string if and only if from any node n of its synchronization graph there is a path to the root.*

Proof. Let us assume that in the synchronization graph there is a path from any node n to ε and let v_n be the binary string formed of codewords on consecutive edges of such a path. The proof is constructive and gives an algorithm for the construction of a synchronizing string by concatenation of the strings v_n.

Let the constructed string s be initialized as ε. We will maintain a set U of nodes that are not brought by s to the root. At start, U is the set of all nodes of the synchronization graph but the root.

At each step i a vertex n_i is picked from U, v_{n_i} is appended to s and U is transformed to $\delta^*(U, v_{n_i}) \setminus \{\varepsilon\}$. When the set U is empty, the string s is a synchronizing string for the Huffman code. This proves one implication of the theorem.

On the other hand, if a code has a synchronizing string s then s is a sequence of codewords. Transitions by consecutive codewords of the string s bring any vertex n of the graph to the root, so it gives a path from n to ε. □

Example 3.8: In the synchronization graph for the code C_2, Figure 3.1(d), for any node there is a path to the root. This means that the code C_2 has a synchronizing string. To construct such a string let us start with the set $U_0 = \{0, 01, 1\}$. Let us pick the node 0 from from U_0. An example path to the root is labeled with b, which corresponds to the binary string 10. The transformed set U is $U_1 = \{0\}$ (0 goes to the root and 01 and 1 go to 0; then the root is removed). The only node in U_1 is 0 and, again, we take the letter b — the string 10. Now the transformation of U_1 with this string results in the empty set. The synchronizing string found is 1010. Note that this is not the shortest synchronizing string as 010 and 011 are shorter.

The time needed to check if there is a synchronizing string for a code is proportional to the size of the synchronization graph, that is $O(N^2)$. Chapter 4 contains a faster algorithm for the same problem.

The construction of a synchronizing string given the proof of Theorem 3.7 requires $O(N^3)$ operations and the constructed string is of length $O(N^3)$ bits. To see that, let us first observe that there is always a path from a node n to ε that consists of at most $N-1$ edges. We can precompute such paths using, for instance, Dijkstra's algorithm. Such path corresponds to a string of source letters of length at most $N-1$ and is equivalent to a binary string of length at most $(N-1)h$. The algorithm performs at most $N-2$ steps, because each step reduces the size of U by one. Each step requires applying a string of letters of length at most $N-1$ to each node in U, which takes $(N-1)|U|$. After summing up all the steps, the total number of operations is bounded by $O(N^3)$.

The synchronizing string is a concatenation of at most $N-2$ strings of length at most $(N-1)h$ each, so its length is $O(N^3)$, as $h = O(N)$. Again, Chapter 4 contains a faster algorithm for the construction of a synchronizing string. Also the synchronizing string constructed by that algorithm is shorter.

3.4 Limited synchronization delay with known start position

In this section we present a modification of Huffman coding to limit the synchronization delay to at most a given number of bits. It is assumed that the decoder starts at an arbitrary bit in the encoded message, but it knows the position of this bit. The redundancy introduced by the method depends on the synchronization properties of the code. It is not required that the code has a synchronizing codeword or a synchronizing string. The method works for any code, but for non-synchronizing codes the introduced redundancy is larger.

The limited synchronization delay will be assured by inserting some additional strings, called *resynchronization markers*, in the encoded message. There are two main design goals for the algorithm:

- The code must remain optimal so that the redundancy depends only on the number of insertions of the resynchronization marker. The redundancy can be made arbitrarily small if the markers are inserted rarely.
- The resynchronization markers are inserted only if necessary. Huffman codes resynchronize spontaneously after a synchronization loss, so, in most cases, explicit insertion of a resynchronization marker is not necessary.

Note that if a suboptimal code is used, the redundancy is nonzero even if no markers are inserted. In such case, the redundancy is roughly proportional to the length of the encoded message.

It is expected that the number of decoders whose synchronization delay would exceed the given threshold is low, therefore, because of the second requirement, few markers are inserted.

Both these requirements are a novelty. Any other method for synchronizing coding uses suboptimal codes [3, 10, 11, 16, 20, 25, 28, 39, 40] or the markers are inserted in

regular intervals and the tendency of Huffman codes to resynchronize spontaneously is not explored [21, 34, 52, 54].

The new method, presented in this section, uses a parameter K, which will be related to the maximum synchronization delay. From now on it is assumed that the decoder starts at a position iK, $i \in \mathbb{N}$ in the encoded message. A decoder that starts at a different place has to skip until the next position iK. It can always be done as the start position of the decoder is known.

The method inserts a resynchronization marker if the synchronization delay of the decoder starting at position iK exceeds K. The decoder will resynchronize after reading the marker. Synchronized decoders are able to detect other marker's insertions and ignore them. The markers are based on the following definition.

Definition 3.9. A *leaf string* for a node n of a Huffman tree, denoted by $L(n)$, is (any) shortest string that brings n to a leaf. An exception for ε is made and it is assumed that $L(\varepsilon) = \epsilon$.

Lemma 3.10. $|L(q)| \leq \lfloor \log N \rfloor$ *for any state* q.

Proof. The size of the subtree of q is at most N, so there must be a leaf that is no farther than $\lfloor \log N \rfloor$ edges away from q. □

The operation of the encoder is the following. At each first codeword boundary at of after position $(i+1)K$, for each $i \in \mathbb{N}$, the encoder computes the current state, q_{iK}, of decoder D_{iK}. If $q_{iK} = \varepsilon$ then D_{iK} is in synchronization and nothing is done. Otherwise, the leaf string for q_{iK} is inserted there, which brings decoder D_{iK} to synchronization.

It has to be assumed that $K \geq h + \lfloor \log N \rfloor$, and then decoder D_{iK} always resynchronizes after at most $K + h + \lfloor \log N \rfloor - 1 < 2K$ bits, counting from position iK, where it started. The encoder has to keep track of the state of each decoder D_{iK} for at most $K + h$ bits and it is done during the encoding of the message. The method will be called knownStartPosition and steps performed by the encoder are presented in Algorithm 3.3.

Example 3.11: Let us consider the following message, $\mathcal{E}$, encoded with code C_1 from Figure 2.3:

$$\mathcal{E} = \underbrace{0\,1}_{b}\,\underbrace{0\,1}_{b}\,\underbrace{1\,1\,1}_{e}\,\underbrace{1\,0\,0}_{a}\,\underbrace{0\,0\,1}_{a}\,\underbrace{1\,1\,0}_{e}\,\underbrace{1\,0\,1}_{b}\,\underbrace{1\,0}_{c}\,\underbrace{1\,1\,1}_{e}\,\underbrace{1\,0}_{c}\,\underbrace{1\,0}_{c}. \tag{3.3}$$

This message will be encoded again with the method knownStartPosition for $K = 6$. In the following equations the positions iK are marked with bullets and with the position number.

The first letters are encoded normally.

$$\mathcal{E}_{(0)} = \underbrace{0\,1}_{b}\ \underbrace{0\,1}_{b}\ \underbrace{1\,1\,\bullet^{6}\,1}_{e} \tag{3.4}$$

Algorithm 3.3: Encoder for the method knownStartPosition.

Input: $\mathcal{M}$ — source message; K — parameter.

```
1  i ← 0;                                  ◁ i is used only for the description of the algorithm
2  p ← 0;          ◁ the current position relative to the bit iK (the position of D_iK)
3  q ← ε;                                  ◁ the state of decoder D_iK at the current position
4  while (a ← M.nextSymbol()) ≠ null do
5      output c(a);                                          ◁ encode the symbol normally
6      q ← δ*(q, c(a));                                      ◁ update the state of D_iK
7      p ← p + |c(a)|;                                       ◁ update the current position of D_iK
8      if p ≥ K then          ◁ make sure D_iK is in sync and start with D_(i+1)K
9          s ← suffix of c(a) of length p − K;               ◁ the overflow from the K-bit span
10         output L(q);                                      ◁ synchronize D_iK; recall that L(ε) = ϵ
11         q ← δ*(ε, sL(q));                                 ◁ the state of D_(i+1)K at the current position
12         p ← |sL(q)|;                                      ◁ the number of bits processed by D_(i+1)K
13         i++;                                              ◁ proceed to the next K-bit span
```

The next letters, until the next iK position, are also encoded normally, but now the encoder follows the state transitions for decoder D_6, that starts at the first bullet.

$$\mathcal{E}_{(1)} = \underbrace{0\,1}_{b}\ \underbrace{0\,1}_{b}\ \underbrace{1\,1\,\bullet^6\,1}_{e}\ \underbrace{0\,0}_{a}\ \underbrace{0\,0}_{a}\ \underbrace{1\,\bullet^{12}\,1\,1}_{e} \tag{3.5}$$

The current position, 14, is the first codeword boundary at or after position 12. Decoder D_6 is now in the state 11, after decoding the codewords 10, 00, 01 and the prefix 11. D_6 is not synchronized, so the leaf string $L(11)$, in this case 0 or 1, has to be inserted.

$$\mathcal{E}_{(2)} = \underbrace{0\,1}_{b}\ \underbrace{0\,1}_{b}\ \underbrace{1\,1\,\bullet^6\,1}_{e}\ \underbrace{0\,0}_{a}\ \underbrace{0\,0}_{a}\ \underbrace{1\,\bullet^{12}\,1\,1}_{e}\ \underbrace{0}_{\times} \tag{3.6}$$

The leaf string is marked with the sign $\times$.

The encoding continues until the next iK boundary.

$$\mathcal{E}_{(3)} = \underbrace{0\,1}_{b}\ \underbrace{0\,1}_{b}\ \underbrace{1\,1\,\bullet^6\,1}_{e}\ \underbrace{0\,0}_{a}\ \underbrace{0\,0}_{a}\ \underbrace{1\,\bullet^{12}\,1\,1}_{e}\ \underbrace{0}_{\times}\ \underbrace{0\,1}_{b}\ \underbrace{1\,\bullet^{18}\,0}_{c} \tag{3.7}$$

At this point, decoder D_{12} is synchronized, so there is no need to insert any leaf string. The encoding continues.

$$\mathcal{E}_{(4)} = \underbrace{0\,1}_{b}\ \underbrace{0\,1}_{b}\ \underbrace{1\,1\,\bullet^6\,1}_{e}\ \underbrace{0\,0}_{a}\ \underbrace{0\,0}_{a}\ \underbrace{1\,\bullet^{12}\,1\,1}_{e}\ \underbrace{0}_{\times}\ \underbrace{0\,1}_{b}\ \underbrace{1\,\bullet^{18}\,0}_{c}\ \underbrace{1\,1\,1}_{e}\ \underbrace{1\,0\,\bullet^{24}}_{c} \tag{3.8}$$

Now, decoder D_{18} is in the state 0, so the leaf string 0 is inserted before proceeding with the encoding. The final message is the following.

$$\mathcal{E}_{(5)} = \underbrace{0\,1}_{b}\ \underbrace{0\,1}_{b}\ \underbrace{1\,1\,\bullet^6\,1}_{e}\ \underbrace{0\,0}_{a}\ \underbrace{0\,0}_{a}\ \underbrace{1\,\bullet^{12}\,1\,1}_{e}\ \underbrace{0}_{\times}\ \underbrace{0\,1}_{b}\ \underbrace{1\,\bullet^{18}\,0}_{c}\ \underbrace{1\,1\,1}_{e}\ \underbrace{1\,0\,\bullet^{24}}_{c}\ \underbrace{0}_{\times}\ \underbrace{1\,0}_{c} \tag{3.9}$$

Algorithm 3.4: Decoding algorithm for the method knownStartPosition.

Input: $\mathcal{E}$ — encoded message; K — parameter.

```
1  i ← (start position)/K;          ◁ i is used only for the description of the algorithm
2  p ← 0;          ◁ the current position relative to the bit iK (the position of D_iK)
3  q ← ε;                          ◁ the state of decoder D_iK at the current position
4  while (a ← E.decodeNextSymbol()) ≠ null do
5      output a;                                          ◁ decode the symbol normally
6      q ← δ*(q, c(a));                                   ◁ update the state of D_iK
7      p ← p + |c(a)|;                                    ◁ update the current position of D_iK
8      if p ≥ K then          ◁ make sure D_iK is in sync and start with D_(i+1)K
9          s ← suffix of c(a) of length p − K;    ◁ the overflow from the K-bit span
10         skip L(q) in E;                  ◁ skip the synchronization marker for D_iK
11         q ← δ*(ε, sL(q));          ◁ the state of D_(i+1)K at the current position
12         p ← |sL(q)|;                ◁ the number of bits processed by D_(i+1)K
13         i++;                                       ◁ proceed to the next K-bit span
```

On decoding a message encoded with knownStartPosition method, the leaf string for decoder D_{jK} has to be skipped by all decoders D_{iK}, $i < j$. On the other hand, decoder D_{jK} must process the leaf string as normal input to be put into synchronization.

It was mentioned before that decoder D_{iK} is always synchronized at position $(i+2)K$. In particular, it knows the correct codewords' boundaries at positions $(i+2)K$, $(i+3)K$ and so on. The decoding algorithm for D_{iK} simulates each decoder D_{jK}, $j > i$, to know whether or not it resynchronizes before the first codeword boundary at or after position $(j+1)K$. At such a position it just compares the state of D_{jk} to its own, because the latter is correct. If the state is different, a leaf string must have been inserted for D_{jK} and it has to be skipped by D_{iK}.

Note that the decoder and the decoding algorithm are distinguished here. The decoder is a Huffman automaton and the decoding algorithm additionally tracks the state of decoders D_{iK} and skips the leaf strings. The decoding algorithm that starts at position p is called the decoding algorithm for D_p. Sometimes, when it does not cause any confusion, the word *decoder* will also be used to refer to the decoding algorithm.

The algorithm for the decoder, Algorithm 3.4, is similar to the one for the encoder. It only differs in lines 1, 4, 5 and 10.

Example 3.12: Let us see how the message from Example 3.11 can be decoded. The message is the following.

$$\mathcal{E}_{(5)} = \underbrace{0\,1}_{b}\underbrace{0\,1}_{b}\underbrace{1\,1\,\bullet^{6}\,1}_{e}\underbrace{0\,0}_{a}\underbrace{0\,0}_{a}\underbrace{1\,\bullet^{12}\,1\,1}_{e}\underbrace{0}_{\times}\underbrace{0\,1}_{b}\underbrace{1\,\bullet^{18}\,0}_{c}\underbrace{1\,1\,1}_{e}\underbrace{1\,0\,\bullet^{24}}_{c}\underbrace{0}_{\times}\underbrace{1\,0}_{c} \tag{3.10}$$

Let us consider the decoding algorithm for D_0 first. It initially decodes the first three letters, *bbe*. Then, it decodes *aae*, but it also computes the state of decoder D_6. At position 14, after D_0 decoded *bbeaae*, decoder D_6 is in

the state 11, because it would decode 10, 00, 10 and the prefix 11. As D_6 is not synchronized at the current position, the decoding algorithm knows that a leaf string for 11 must have been inserted by the encoded. The next bit, the leaf string 0, is skipped. Then decoding continues and D_0 decodes *bc*. At the current position, 19, decoder D_{12} is synchronized, so no leaf string has been inserted at this position. The decoding algorithm continues with decoding *ec*. Now, at position 24, decoder D_{18} is not synchronized, so the leaf string 0 for the state 0 is skipped. Finally, the letter *c* is decoded.

Let us consider now the decoding algorithm for D_6, which starts at the first bullet. D_6 is not synchronized at start. First, it decodes the bits 10, 00, 01 and 110 as the string *cabd*. Note that the last codeword 110 included the leaf string 0 as the last bit. Now the decoder is synchronized and the decoding algorithm continues with decoding *bc*. At position 19 the decoding algorithm crosses a 6-bit block boundary, so it has to check if decoder D_{12} is synchronized. Note that D_6 is synchronized at this position, so the correct state is known. Moreover, the decoding algorithm for D_6 can access all the bits that are read by D_{12}.

The decoding algorithm verifies that D_{12} is synchronized at position 19, so the decoding continues and *ec* is decoded. Now, decoder D_{18} is not synchronized, so the leaf string 0 is skipped and finally *c* is decoded. The string decoded by the decoder is *cabdbcecc*. The four initial characters are wrong, because the decoder was not synchronized when they were decoded.

Decoder D_{12}, starting at the second bullet, decodes first the codeword 110 as *d*. Then it is synchronized and decodes *bcec*. Afterwards, it skips the leaf string 0, inserted for decoder D_{18}, and decodes *c*.

Theorem 3.13. *The redundancy introduced by Algorithm 3.3 is at most* $\frac{\lfloor \log N \rfloor}{K - \lfloor \log N \rfloor}$ *bits per bit of the unmodified encoded message. The redundancy is zero if all decoders starting at positions* iK, $i \geq 0$, *synchronize in at most* K *bits.*

Proof. By Theorem 3.10, the number of inserted bits is at most $\lfloor \log N \rfloor$ for every K bits. The remaining bits, at least $K - \lfloor \log N \rfloor$, are the bits of the unmodified encoded message.

If any decoder starting at positions iK, $i \geq 0$, synchronize in at most K bits, no leaf string is ever inserted, so the redundancy is zero. □

Note that even though the redundancy is bounded by $\frac{\lfloor \log N \rfloor}{K - \lfloor \log N \rfloor}$ in the worst case, the redundancy is much smaller if the code has good statistical synchronization capabilities, in particular, if K is much larger than the average synchronization delay of the code.

Example 3.14: The redundancy in the message from Example 3.12 is 2 bits and length of the original message is 25 bits. This gives the redundancy of 0.08 bits per bit of the unmodified encoded message. The bound from Theorem 3.13 is $\frac{\lfloor \log 5 \rfloor}{6 - \lfloor \log 5 \rfloor} = \frac{2}{6-2} = 0.5$. Note that in typical applications the value of K is much larger than $\log N$, so the bound is smaller.

For the computation of the complexity for the algorithm we consider only its overhead over normal Huffman coding, therefore the cost of line 5 is considered to be zero. The speed of the algorithm (both the encoder and the decoder) depends on the time needed for executing line 6, that is for the instruction $q \leftarrow \delta^*(q, c(a))$. If the function δ^* is precomputed for all the (q, a) pairs, the time is constant. In this case the overhead over normal Huffman coding is $O(R + |\mathcal{M}|)$, where R is the number of inserted bits, and it is sublinear in the length of the encoded message. The preprocessing, Algorithm 3.2, requires $O(N^2)$ time.

The preprocessing time, $O(N^2)$, is negligible only if the code is small in comparison to $|\mathcal{E}|$. If, however, the size of the code is $O(\sqrt{|\mathcal{E}|})$, it is preferable to reduce the precomputation time, even at the cost of increased processing complexity. Without preprocessing the computation in line 6 requires $O(|c(a|))$ operations and the total time overhead of Algorithm 3.3 over normal Huffman coding is $O(R + |\mathcal{E}|)$, which is linear in the length of the encoded message.

In both cases this is the worst-case complexity. Both the encoder and the decoder may be optimized. Indeed, line 6 does not have to be executed if the state q is ε.

The resynchronization properties of the method can be summarized as follows:

Theorem 3.15. *The decoder that starts at a position iK of a message encoded with Algorithm 3.3 and that executes Algorithm 3.4 resynchronizes after at most $K + h - 1 + \lfloor \log N \rfloor$ bits.*

Proof. Let us look at the decoder at the first codeword boundary at or after position $(i+1)K$. This has to be before position $K+h-1$, because the longest codeword is of length h. If D_{iK} is synchronized then the theorem is proved. Otherwise, the encoder must have inserted a leaf string for the current state of D_{iK}. The length of the leaf string, by Theorem 3.10, is at most $\lfloor \log N \rfloor$ bits and the decoder is synchronized upon reading these bits. □

Example 3.16: Let us look at the message from Example 3.12. Decoder D_0 synchronized immediately, decoder D_6 after 9 bits, D_{12} after 3 bits and D_{18} after 7 bits. The bound from Theorem 3.15 is $6 + 3 - 1 + 2 = 10$. In fact, for the code C_1 no leaf string is longer than 1, so the bound can be improved to 9.

Theorem 3.17. *A decoding algorithm that starts at a known position p of a message encoded with Algorithm 3.3 resynchronizes after at most $2K + h - 2 + \lfloor \log N \rfloor$ bits.*

Proof. The decoder first skips ($p \mod K$) bits, which is at most $K - 1$. Then, it is at a position iK and may execute Algorithm 3.4. By Theorem 3.15, the decoder synchronizes after the next $K + h - 1 + \lfloor \log N \rfloor$ bits, which proves the bound. □

An important application of the method is limiting the propagation of bit-flip errors. Let us consider an error at position p with $iK \leq p < (i + 1)K$. Any decoder that started before p may be corrupted. Decoder $D_{(i+1)K}$ is not influenced by the error and it resynchronizes in $K + h - 1 + \lfloor \log N \rfloor$ bits. Decoder D_{iK} can then adjust its state at position $(i+2)K + h - 1 + \lfloor \log N \rfloor$ to be the same as the state of $D_{(i+1)K}$. Therefore, the error propagates for at most $2K + h - 1 + \lfloor \log N \rfloor$ bits. Note that

Position	RESYNCHRONIZED	UNSYNCHRONIZED
0	0	NULL
2	2	1
5	5	1
7	1	3
9	9	3

Table 3.1: The information about synchronized and unsynchronized decoders

the method cannot be used when bit insertion or bit deletion errors occur, because it assumes that the decoder always knows its exact position in $\mathcal{E}$.

3.5 Tracking decoders

In this section it is assumed that the whole source message, $\mathcal{M}$, and its encoding, $\mathcal{E}$, are available. Let us consider the problem of finding at each position p that is a codeword boundary the minimal start position of a decoder that:

- resynchronizes at p — this piece of information is called RESYNCHRONIZED(p),
- is not synchronized at p — called UNSYNCHRONIZED(p).

This information can be used, for instance, to compute the largest synchronization delay of any decoder or to limit the synchronization delay by inserting some resynchronization markers at positions where the synchronization delay of some decoder is too large. The latter application will be discussed in Section 3.6.

Example 3.18: Let us consider the following message, $\mathcal{E}$, encoded with the code C_1.

$$\mathcal{E} = {}^{0}\ \underbrace{0\,1}_{b}\ {}^{2}\ \underbrace{1\,1\,1}_{e}\ {}^{5}\ \underbrace{1\,0}_{c}\ {}^{7}\ \underbrace{0\,1}_{b}\ {}^{9}. \tag{3.11}$$

The positions between codewords are numbered. The information we are looking for is the following. At position 0 the decoder that resynchronizes is decoder D_0. There is no decoder that is unsynchronized at position 0. At position 2, decoder D_2 resynchronizes, decoder D_1 is unsynchronized and decoder D_0 was already synchronized at position 0. The minimal start position of a decoder that is unsynchronized is 1, and of a decoder that resynchronizes is 2.

The information at other positions is presented in Table 3.1. The difference of columns "RESYNCHRONIZED" and "Position" gives the synchronization delay for a decoder that resynchronized at the given position. The largest synchronization delay is 6 for decoder D_1. Note that decoder D_3 is not synchronized at the end of the message.

Two methods for computing RESYNCHRONIZED(p) and UNSYNCHRONIZED(p) for all codeword boundaries p are presented in the next subsections. The first one is very

simple, yet, in most cases, efficient. It uses $O(h)$ operations per encoded codeword. The other one is asymptotically optimal, as an algorithm that works on the encoded message, because the amortized amount of computations per encoded bit is constant.

3.5.1 O(h) per codeword

To find the values of RESYNCHRONIZED and UNSYNCHRONIZED at each position, each decoder of $\mathcal{E}$ has to be analyzed to find the exact place where it resynchronizes (and not only the upper bound, as in Section 3.2). Let us consider decoder D_p. Position p is inside of some encoded codeword w_0 and it determines some proper suffix, possibly empty, w_0' of w_0. We will look into state transitions of the Huffman automaton D_p. At the end of w_0 decoder D_p is in the state $q_0 = \delta^*(\varepsilon, w_0')$. If $q_0 \neq \varepsilon$ the decoder is not synchronized. After each next codeword w_i the decoder moves to $q_i = \delta^*(q_{i-1}, w_i)$.

Recall that the correct decoder is always in the state ε on codeword boundaries. Note also that ε is the only state of the Huffman automaton where states of two decoders can merge. It follows that the first index i such that $q_i = \varepsilon$ determines exactly the resynchronization position for D_p (this may also happen for q_0).

This observation gives an algorithm for determining the synchronization delay for a decoder starting at a particular position in $\mathcal{E}$. To compute it for all decoders, instead of making several passes through the message, it is preferable to analyze the synchronization delays while encoding $\mathcal{M}$, with just one pass.

At each position p that is a codeword boundary we maintain the set S of decoders that started before p and are not synchronized at p. For each such decoder its state at p is kept. After encoding a codeword w_i, each decoder of the set S makes a transition from its state, q, to $q' = \delta^*(q, w_i)$. If q' happens to be the root, the decoder resynchronizes and is removed from S. The information about the resynchronization and also about the decoder with the largest synchronization delay so far is memorized. Next, all $|w_i| - 1$ decoders that start inside w_i and that are unsynchronized at the end of it are added to S.

It is enough to keep in S only decoders that are in different states. Indeed, if D_{p_1} and D_{p_2}, $p_1 < p_2$, are at position p in a common state q, their state always remains common. It follows that they resynchronize at the same position, so it is enough to consider only the decoder with larger synchronization delay, namely D_{p_1}.

Example 3.19: Let us consider the message $\mathcal{E}$ from Example 3.18 encoded with the code C_1.

$$\mathcal{E} = {}^0\ \underbrace{0\,1}_{b}\ {}^2\ \underbrace{1\,1\,1}_{e}\ {}^5\ \underbrace{1\,0}_{c}\ {}^7\ \underbrace{0\,1}_{b}\ {}^9. \tag{3.12}$$

The contents of the set S on codewords boundaries is presented in Table 3.2. The set S is initially empty. At position 2, decoder D_1 in the state 1 is added. At position 5, decoder D_1 proceeds to the state $\delta^*(1, 111) = 1$. Then, decoders D_3 and D_4 in states 11 and 1, respectively, are added to S. But D_4 is in the state 1, the same as D_1, so it is removed immediately. At position 7 decoder D_1 resynchronizes and is removed from S. Decoder D_3 proceeds to the state 0. Decoder D_6 is not added to S because its state is the same as the state of

Position	The set S: decoder (state)
0	$\emptyset$
2	D_1 (1)
5	D_1 (1), D_3 (11)
7	D_3 (0)
9	D_3 (1)

Table 3.2: The set S at codeword boundaries for the message from Example 3.18.

D_3 and D_3 started earlier. At position 9 decoder D_3 moves to 1 and D_8 is not added, because its state is the same as the state of D_3, which is already in S.

Algorithm 3.5 formalizes the ideas described above. The set S stores *(state, position)* pairs, where *position* identifies a decoder and *state* is the state of this decoder at the current position. It is assumed that the operation $S.\text{ADD}(q, p_0)$ checks if there is another (q, p_0') already present in S and, if so, always leaves in S only $(q, \min(p_0, p_0'))$ of the two.

The set S is initially empty. In the loop of line 4, consecutive letters of $\mathcal{M}$ are analyzed. This involves taking a decoder D_{p_0} out of the set S and computing its next state, after processing the current letter. If the state is ε, the decoder resynchronizes. Otherwise, it is added to S' — the working copy of S.

After all decoders from S have been processed, new decoders, that started inside the current codeword, are added in the loop in line 16. The list SUFFIXES(w), for a codeword w, contains decoders of w that start inside of w and are not synchronized at the end of w. As in the set S, with the state of a decoder we store also its start position, relative to the start of the codeword w. If there are two decoders in the same state at the end of w, only the decoder that started earlier is kept. In addition, the values MINSYNCHRONIZED(w) and MINUNSYNCHRONIZED(w), used in lines 18 and 19, contain the minimal start position of a decoder that starts inside of w and, respectively, is synchronized and unsynchronized at the end of w.

Example 3.20: Table 3.3 contains the list SUFFIXES and the values of MINSYNCHRONIZED and MINUNSYNCHRONIZED for the code C_1.

The information being computed by Algorithm 3.5, that is the values of RESYNCHRONIZED and UNSYNCHRONIZED at the current position, is memorized in lines 21 and 22.

The time complexity for the algorithm depends on the size of the set S. It is easy to see that $|S| < N - 1$, because all the decoders in S are in different states. This bound can be improved using the following Lemma.

Lemma 3.21. *If D_{p_0} is in state q at position p in $\mathcal{E}$ then $\pi(q)$ is a suffix of $\mathcal{E}[..p)$.*

Proof. At position p decoder D_{p_0} is in the state $q = \delta^*(\varepsilon, \mathcal{E}[p_0..p))$. Let t be the longest prefix of $\mathcal{E}[p_0..p)$ such that $\delta^*(\varepsilon, t) = \varepsilon$, and let $\mathcal{E}[p_0..p) = tt'$. Then $t' = \pi(q)$ and it is a suffix of $\mathcal{E}[p_0..p)$ and also of $\mathcal{E}[..p)$. □

Algorithm 3.5: Decoder tracking for computing the synchronization delay.

Input: $\mathcal{M}$ — source message.
Output: The values of RESYNCHRONIZED(p) and UNSYNCHRONIZED(p) at each codeword boundary p.

```
1  S ← ∅;                                          ◁ the set of unsynchronized decoders
2  S' ← ∅;                                         ◁ a working copy of the set S
3  p ← 0;                                          ◁ the current position in the ℰ
4  foreach letter l in ℳ do
5      w ← c(l);                                   ◁ the codeword for l
6      s_s ← +∞;   ◁ minimal start position of a synchronized decoder (temporary)
7      s_u ← +∞;   ◁ minimal start position of an unsynchronized decoder (temporary)
8      while S ≠ ∅ do                              ◁ process all decoders from S
9          (q, p_0) ← S.REMOVEANY();               ◁ get a decoder
10         q' ← δ*(q, w);                          ◁ compute its new state
11         if q' ≠ ε then                          ◁ not synchronized
12             S'.ADD(q', p_0);
13             s_u ← MIN(s_u, p_0);
14         else                                    ◁ resynchronized
15             s_s ← MIN(s_s, p_0);
16     foreach (q, n) ∈ SUFFIXES(w) do ◁ add new decoders that started inside w
17         S'.ADD(q, p + n);
18     s_u ← MIN(s_u, p + MINUNSYNCHRONIZED(w));
19     s_s ← MIN(s_s, p + MINSYNCHRONIZED(w));
20     p ← p + |w|;                                ◁ advance to the next codeword
21     if s_s < +∞ then RESYNCHRONIZED(p) ← s_s;
22     if s_u < +∞ then UNSYNCHRONIZED(p) ← s_u;
23     (S, S') ← (S', ∅);          ◁ swap the sets S and S' (S is already empty)
```

Codeword	SUFFIXES	MINSYNCHRONIZED	MINUNSYNCHRONIZED
00	D_1 (0)	2	1
01	D_1 (1)	2	1
10	D_1 (0)	2	1
110	D_2 (0)	1	2
111	D_1 (11), D_2 (1)	3	1

Table 3.3: The values SUFFIXES, MINSYNCHRONIZED and MINUNSYNCHRONIZED for the code C_1.

Algorithm 3.6: Computation of the lists SUFFIXES.

Input: T — Huffman tree.
Output: The contents of the list SUFFIXES(w) for each codeword w.

```
1  procedure SearchSuffixes(x, y, k);
2    if x is a leaf then
3      SearchSuffixes(ε, y, k);
4    else
5      if y is a leaf then
6        suffixes(π(y)).add(x, k);
7      else
8        SearchSuffixes(x.left, y.left, k + 1);
9        SearchSuffixes(x.right, y.right, k + 1);
10 end procedure;
11 forall internal node v ∈ T do
12   SearchSuffixes(ε, v, 0);
```

Example 3.22: Let us look at the message $\mathcal{E}$ from Example 3.18, encoded with the code C_1. At position 3 decoder D_1 is in the state 11, which is indeed a suffix of 011 — the prefix of $\mathcal{E}$ of length 3. Similarly, at position 7 decoder D_5 is in the state ε. Trivially, ε is a suffix of 0111110.

Lemma 3.23. *The size of the set S is less or equal h.*

Proof. It is enough to prove that all the decoders from S are in states of different distance from the root, because the maximum distance from the root is $h - 1$. This is indeed the case. If there were two decoders in S in states q_1, q_2, with $|\pi(q_1)| = |\pi(q_2)|$ then, by Lemma 3.21, $\pi(q_1)$ and $\pi(q_2)$ are two suffixes of $\mathcal{E}[..p)$. But they are of the same length so they must be equal and $q_1 = q_2$. □

Example 3.24: It is easy to see that the size of the set S from Table 3.2, for the message $\mathcal{E}$ from Example 3.18, is never larger than 3.

Efficient implementation of Algorithm 3.5 requires precomputing the values of $\delta^*(q, w)$ (line 10) for any state q and any codeword w. This is equivalent to the synchronization graph (Section 3.3) and requires $O(N^2)$ operations (Algorithm 3.2).

It remains to be shown how to compute the lists SUFFIXES and the values MINSYNCHRONIZED and MINUNSYNCHRONIZED. In a naïve approach, the lists SUFFIXES can be computed for each codeword w_i by processing all its suffixes. Processing a suffix, v, requires $O(|v|)$ operations, so the total processing time is $O(\sum |w_i|^2) = O(N^3)$.

Algorithm 3.6 computes the lists SUFFIXES with fewer operations. It achieves its $O(\sum |w_i|)$ time complexity by modifying the order in which the suffixes are analyzed, and by reusing some computations. The main loop of the algorithm goes over all internal nodes of the Huffman tree. Each node corresponds to the set of codewords' suffixes that start at that node. Algorithm 3.6 analyses all such suffixes at one stroke.

Algorithm 3.7: Computation of the lists SUFFIXES — optimized version.

Input: T — Huffman tree.
Output: The contents of the list SUFFIXES(w) for each codeword w.

```
1  procedure SearchSuffixes(x, y, k);
2    if x is a leaf then
3      N[y] ← max(N[y], k);
4    else
5      if y is a leaf then
6        suffixes(π(y)).add(x, k);
7      else
8        SearchSuffixes(x.left, y.left, k + 1);
9        SearchSuffixes(x.right, y.right, k + 1);
10 end procedure;
11 forall internal node v ∈ T do
12   N[v] ← 0;
13 forall internal node v ∈ T in BFS order do
14   SearchSuffixes(ε, v, N[v]);
```

Processing an internal node, n, is analogous to Algorithm 3.2 — computations for common prefixes of the suffixes starting at n are reused. It is assumed that the operation ADD(q, n) has the same properties as in Algorithm 3.5 — it checks if there is another (q, n') already present in the set and, if so, always leaves in the set only $(q, \min(n, n'))$ of the two.

The number of operations needed to process a single internal node, n, is proportional to the size of the subtree rooted at n. The total cost of this algorithm for a tree T is proportional to the sum of sizes of all subtrees of T, denoted S_T, which is, by Lemma 4.2, $O(\sum |w_i|)$.

The property of the operation ADD can be used to introduce yet another optimization to Algorithm 3.6. Let us consider a suffix v' of some codeword v. The word v' may be either a prefix of some codeword or $v' = wv''$, where w is a codeword. In the first case, the suffix v' is processed normally, as in Algorithm 3.6. In the latter case, the decoder that processes either wv'' or v'' finishes in the same state q, so it is enough to process v'' only once. The processing of $v' = wv''$ may stop after w and the length of w may be stored for later use. In the set SUFFIXES(w) only the pair $(q, |v| - |v'|) = (q, |v| - |v''| - |w|)$ is kept, because it corresponds to a lower start position than the pair $(q, |v| - |v''|)$. When the suffix v'' is processed, the value $|w|$, memorized earlier, is subtracted from $|v| - |v''|$, the start position of the decoder that processes v'', to get the value $|v| - |v''| - |w|$.

The optimized version of Algorithm 3.6 is presented in Algorithm 3.7. The information about the processed prefixes of suffixes of codewords is kept in the array $N[v]$, indexed with internal nodes v of the Huffman tree. When the function SEARCHSUFFIXES is called from the main loop with the node v as its second parameter, the value

of $N[v]$ is the length of the longest suffix of $\pi(v)$ that is a sequence of codewords. The array $N[v]$ is initialized to 0 at start. Despite the additional optimization, the time complexity for Algorithm 3.7 is still $O(\sum |w_i|)$.

Modifications of Algorithms 3.6 and 3.7 for the computation of the values minSynchronized and minUnsynchronized are left to the reader. The complexity for Algorithm 3.5 is summarized in the following theorem:

Theorem 3.25. *Algorithm 3.5 requires $O(h)$ operations per encoded codeword with $O(N^2)$ preprocessing time and $O(N^2)$ preprocessing memory.*

Note that this is the worst-case setting. On average the size of the set S will be smaller than h, especially for codes with unbalanced Huffman trees.

Example 3.26: Let us consider the message $\mathcal{E}$ from Example 3.18, encoded with the code C_1 from Figure 2.3, to see how Algorithm 3.5 works. First the set S is empty. The first letter of the message is b, which corresponds to 01. The decoders from the list suffixes(01) are added to S with their position increased by the value of the current position, which is 0. Therefore the set S contains only the pair (1, 1), which is decoder D_1 in the state 1. The value of resynchronized is equal 2 and of unsynchronized — 1. These are the values of minSynchronized and minUnsynchronized for the codeword 01.

After processing the next codeword, 111, the decoder D_1 from S makes a transition to $\delta^*(1, 111) = 1$. Now, the set S contains the pair (11, 1). The minimal start position of an unsynchronized decoder, s_u, is 1. Then, the decoders of suffixes(111) are added to S, with their position increased by 2. In fact, the decoder that starts from the last bit of 111 is skipped, because it is in the same state as D_1, which is already in S. For the other decoder, the pair (11, 3) is added to S. The value of s_u is not changed. The minimal value of an synchronized decoder is $2 + \text{minSynchronized}(111) = 2 + 3 = 5$ (compare with Table 3.1).

The next codeword is 10. The decoder D_1 from S moves to the state ε and is removed. The minimal position of a resynchronized decoder is set to 1. Next, the decoder D_3 from S moves to 0. The only decoder on the list suffixes(10) is the decoder that starts from the second bit. Its target state is also 0, so it is not added to S. The minimal start position of a resynchronized decoder is equal 1 and for an unsynchronized decoder is equal 3.

The last codeword is 01. Decoder D_3 moves to the state 1. The only decoder of the list suffixes(01) is also in the state 1, so it is ignored. No decoder resynchronizes apart from the one that starts at position 9. The decoder with the largest synchronization delay is D_3, the only decoder that remains in S.

3.5.2 O(1) per bit

There are two drawbacks of using Algorithm 3.5 for tracking decoders. Firstly, the preprocessing time is quadratic in the size of the code. For small codes and long

Node	BORD	LEAF	NONLEAF
ε	NULL	NULL	NULL
0	ε	NULL	ε
1	ε	NULL	ε
00	0	NULL	0
01	1	NULL	1
10	0	NULL	0
11	1	NULL	1
110	10	10	0
111	11	NULL	11

Table 3.4: The values BORD, LEAF and NONLEAF for the code C_1.

encoded messages the preprocessing time is negligible, but if the code is large, the preprocessing may dominate the total execution time, degrading the performance. As will soon be shown, it is possible to achieve the goals of Algorithm 3.5 using a different approach. The new algorithm tracks all decoders in $O(1)$ amortized time per encoded bit. Unlike Algorithm 3.5, the new algorithm processes codewords bit by bit and does not use the synchronization graph. The preprocessing is reduced to linear time in the size of the code.

The following information is needed for each node, v, of the Huffman tree for the code:

- v.BORD — the lowest node v' such that $\pi(v')$ is a proper suffix of $\pi(v)$,
- v.LEAF — the lowest leaf node v' such that $\pi(v')$ is a proper suffix of $\pi(v)$,
- v.NONLEAF — the lowest non-leaf node v' such that $\pi(v')$ is a proper suffix of $\pi(v)$.

By the lowest node it is meant the node with the largest distance from the root. If no node fulfilling the properties exists, the value is NULL.

Example 3.27: The values BORD, LEAF and NONLEAF for the code C_1 from Figure 2.3 are shown in Table 3.4.

Algorithm 3.8 computes the values of BORD, LEAF and NONLEAF for all nodes of a given Huffman tree. It first initializes the values at the root. Then, it computes the values at each node v using values for v's predecessors, which were computed before v is considered. As the processing time for each vertex is constant, the algorithm works in $O(N)$ time.

The new algorithm for tracking decoders is Algorithm 3.9. As before, the algorithm maintains the set S of states of unsynchronized decoders that process $\mathcal{E}$, but this time the set S is updated at each position in $\mathcal{E}$, an not only at codeword boundaries. We keep the restriction that of two decoders in the same state, only the one with lower start position is stored.

Algorithm 3.8: Computation of the fields BORD, LEAF and NONLEAF.

Input: T — Huffman tree.
Output: The fields BORD, LEAF and NONLEAF for each node v of T.

```
1  ε.BORD ← NULL;
2  ε.LEAF ← NULL;
3  ε.NONLEAF ← NULL;
4  foreach node v of T, other than ε, in BFS order do
5      t ← v.PARENT;
6      z ← t.NONLEAF;
7      b ← bit such that v is the b-child of t;
8      if z ≠ NULL then
9          v.BORD ← δ_T(z, b);
10     else
11         v.BORD ← ε;
12     if v.BORD is a leaf then
13         v.NONLEAF ← v.BORD.NONLEAF;
14         v.LEAF ← v.BORD;
15     else
16         v.NONLEAF ← v.BORD;
17         v.LEAF ← v.BORD.LEAF;
```

Example 3.28: The set S of unsynchronized decoders at each position in the message from Example 3.18 is shown in Table 3.5. Its construction is described below.

The set S is initially empty. At position 1 decoder D_1 in the state ε is added. At position 2 decoder D_1 proceeds to the state 1 and no new decoders are added. At position 3 decoder D_1 proceeds to the state 11 and decoder D_3 is added with the state ε. At position 4 decoder D_1 moves to ε and decoder D_3 moves to 1. Decoder D_4 is not added to the set, because decoder D_1 in the state ε is already present there. This procedure continues until the last bit of the message.

By Lemma 3.21, a decoder in a state q corresponds to the suffix $\pi(q)$ of $\mathcal{E}[0..p)$. The longest such suffix, $\pi(q_l)$, determines the possible states of all other decoders at position p. Their states correspond to suffixes w of $\pi(q_l)$ such that w brings ε to $\delta^*(\varepsilon, w)$ without passing through a leaf, formally: $\delta^*(\varepsilon, w') \neq \varepsilon$ for any nonempty prefix w' of w. Hence, all the states are in the set

$$\{q_l, q_l.\text{NONLEAF}, q_l.\text{NONLEAF.NONLEAF}, \ldots\}. \tag{3.13}$$

After a transition by a bit b, the state q_l becomes $q_l' = \delta(q_l, b)$ if it is not a leaf, or it becomes $q_l' = \delta(q_l, b).\text{NONLEAF}$ otherwise.

Example 3.29: At position 5 in the message from Example 3.18 the state $q_l = 11$ of the decoder D_3, is of the largest distance from the root. The set

$$\{q_l, q_l.\text{NONLEAF}, q_l.\text{NONLEAF.NONLEAF}, \ldots\}$$

Algorithm 3.9: Decoder tracking in $O(1)$ per bit.

Input: $\mathcal{E}$ — encoded message.
Output: The values of RESYNCHRONIZED(p) and UNSYNCHRONIZED(p) at each codeword boundary p.

```
1  p ← 0;
2  q_l ← ε;
3  foreach letter l of M do
4      w ← c(l);
5      foreach bit b in w do
6          p++;
7          q'_l ← δ_T(q_l, b);
8          Shift the array A by one;
9          if q'_l is a leaf then
10             q_l ← q'_l.nonLeaf;
11         else
12             q_l ← q'_l;
13             q'_l ← q'_l.leaf;
14         x ← the state q ∈ {q'_l, q'_l.leaf, q'_l.leaf.leaf, ...} with max. A[|π(q)|];
15         Delete all (q'_l, q'_l.leaf, q'_l.leaf.leaf, ...) from A and L but x;
16         if x ≠ null then
17             move x from A[|π(x)|] to A[0];
18         else
19             insert decoder D_p at A[0] and at the head of L;
20     resynchronized(p) ← A[0];
21     delete A[0] from A and L;
22     unsynchronized(p) ← the last element of L;
```

Position	The set S: decoder (state)
0	
1	D_1 (ε)
2	D_1 (1)
3	D_1 (11), D_3 (ε)
4	D_1 (ε), D_3 (1)
5	D_1 (1), D_3 (11)
6	D_1 (11), D_3 (ε)
7	D_3 (0)
8	D_3 (ε)
9	D_3 (1)

Table 3.5: The set S for the message from Example 3.18.

position	q_l	$A[0]$	$A[1]$	$A[2]$
0	ε	NULL	NULL	NULL
1	0	D_1	NULL	NULL
2	1	NULL	D_1	NULL
3	11	D_3	NULL	D_1
4	11	D_1	D_3	NULL
5	11	NULL	D_1	D_3
6	11	D_3	NULL	D_1
7	0	NULL	D_3	NULL
8	0	D_3	NULL	NULL
9	1	NULL	D_3	NULL

Table 3.6: The array A from Example 3.30.

consists of the states 11, 1 and ε and gives all possible states of other decoders in S. Indeed, decoder D_1 is in the state 1 and D_3 is in 11. The decoder in the state ε is synchronized at that position, so it is not in S.

Similarly, at position 6 the decoder with the lowest state is D_1 in the state 11. Now the decoder in the state 1 is synchronized, so it is not present in S. The decoder in the state ε is decoder D_3.

This representation of decoders' states is used to maintain the set S of unsynchronized decoders. The start position for a decoder being in a state q is stored in an array $A[0..h-1]$ of integers, at index $|\pi(q)|$. The value $A[i]$ is NULL if the suffix of $\mathcal{E}$ of length i does not correspond to any decoder's state, or if such a decoder is synchronized.

Example 3.30: Table 3.6 shows how the array A changes while processing the message from Example 3.18. Compare these values with Tables 3.1 and 3.5.

Let us consider decoder D_p to see how processing a single bit b influences this representation of S. Let q be the state of D_p before processing the bit b. It follows that $A[|\pi(q)|] = p$. After processing b, the state turns into $\delta(q, b)$ if it is not a leaf, or into ε otherwise. In the first case $A[|\pi(q)| + 1]$ is set to p, which means that the contents of A is shifted by one element. In the second case the value of $A[|\pi(q)| + 1]$ is set to NULL and p is put into $A[0]$. In the latter case the state of D_p is ε, so if the current position is a codeword boundary, D_p is removed from S and $A[0]$ is set to NULL.

To apply the changes to all decoders, first the array A has to be shifted by one. This can be done in constant time by adding an cyclic-shift offset at each access to A. Then, all decoders that ended up in a leaf have to be removed from A. Let $q'_l = \delta(q_l, b)$. The decoders that appeared in leaves are q'_l.LEAF, q'_l.LEAF.LEAF, ..., plus additionally q'_l if it is a leaf itself. The removed decoders are now in the state ε, so they should appear at $A[0]$. The final value of $A[0]$ is the minimum start position of the removed decoders, if there were any. Otherwise, it is the current position (after the bit b). Finally, if the current position is the codeword boundary, the value of $A[0]$

is set to NULL, which means that the corresponding decoder resynchronized.

Example 3.31: Let us analyze how array A and the state q_l in Example 3.30 change while encoding consecutive bits.

At start, q_l is equal ε. At position 1, after the first bit, 0, the state q_l changes to 0. The value of q_l.LEAF is NULL, so no decoders are moved to the root. Position 1 is not a codeword boundary, so decoder D_1 is added to A at the index 0.

The second bit forces the transition of q_l by the bit 1. The result state, 01, is a leaf, so the next value of q_l is 01.NONLEAF, which is equal 1. The array A is shifted by one, so decoder D_1 ends up in $A[1]$.

Next, the bit 1 moves q_l to 11. The decoder from $A[1]$ moves to $A[2]$ and a new decoder, D_3, is added at $A[0]$.

At position 4, the state q_l changes to 111, which is a leaf, so it is replaced by 111.NONLEAF = 11. Array A has to be shifted, so decoder D_1 moves to $A[3]$ and D_3 to $A[1]$. The decoder from $A[|111|]$, D_1, (and only this one, because 111.LEAF is NULL) is moved to $A[0]$. No decoder is added in $A[0]$, because decoder D_1 is already there.

We skip the description of changes after the fifth and sixth bit. The changes at position 7 are similar to the ones at position 4, because decoder D_1 moves to a leaf and is placed in $A[0]$. But position 7 is a codeword boundary, so the decoder is removed from $A[0]$ afterwards.

Lemma 3.32. *The amortized cost of updating A after encoding a bit is constant.*

Proof. The proof utilizes the credit method. At each position two credits are used. The first one is spent for adding a new decoder at $A[0]$. The other one is given to the decoder at $A[0]$. In this way all the decoders in A always have one credit. The credit is used when the decoder moves to a leaf and is removed. □

With this representation of the set S the problem of finding in S the decoder with the largest synchronization delay in constant time is not trivial. For this purpose we maintain a double-linked list, L, with all the decoders from S, sorted by their start position. The array A is changed to keep elements of that list instead of just the start positions of decoders. Thus, elements of the array A are structures containing the start position of the decoder and pointers to the previous (larger p) and the next (smaller p) structure on the list L. The shift of A works as before. In order to remove decoders that moved into a leaf, the one of them with the lowest start position, $D_{p'}$, has to be found first, using A. Then, all other decoders are removed from both L and A. Finally, the structure for $D_{p'}$ is moved to $A[0]$ without changing its position in L. This guarantees that L remains sorted. If there were no decoders that moved to a leaf, the structure for the new decoder at $A[0]$ is placed at the head of the list L.

Theorem 3.33. *The amortized cost of Algorithm 3.9 is $O(1)$ per analyzed bit.*

Proof. By Lemma 3.32, the amortized cost of line 15 is constant. The cost of the line 14 can be included in the cost of line 15, because it requires proportional amount of time. The rest of operations require constant time per bit. □

3.6 Limited synchronization delay with unknown start position

This section contains another method for limiting the synchronization delay of a decoder that starts at an arbitrary position in the encoded message. The two design goals stated in Section 3.4 are still of our interest, but this time the assumption that the decoder knows its position in the encoded message is released.

Let us assume for a moment that the code used for encoding has a synchronizing codeword, w_s, and that w_s is ignored by decoders (it is decoded to an empty character). The string w_s will be used as a *resynchronization marker*, denoted by $\mathcal{R}$.

It is important to note that the assumptions that $\mathcal{R}$ is a codeword that is decoded to an empty character implies that the code is not optimal. However, this assumption is not necessary and is introduced only to simplify the presentation. A method to avoid it will soon be discussed. Moreover, as will soon be shown, the method works for all synchronizing Huffman codes, and not only for the ones with a synchronizing codeword.

The limited synchronization delay can be provided by inserting $\mathcal{R}$ at some positions in $\mathcal{E}$, between two codewords. Each decoder that started before an inserted synchronizing codeword synchronizes at the end of it. It further ignores all occurrences of the marker and thus decodes the same letters as in the source message $\mathcal{M}$.

A naïve approach to provide limited synchronization delay is to insert the marker in regular intervals, each time the number of bits from the beginning of the last insertion of $\mathcal{R}$ exceeds K, where K is a parameter. Then, each decoder that started at the second bit of $\mathcal{R}$ or farther resynchronizes in the worst case at the next insertion of the marker, that is no later than after $K + h + |\mathcal{R}|$ bits (h is added because markers are inserted only between codewords and the last codeword before the K-th bit may exceed the K bits boundary by at most h). The redundancy introduced by such insertions of the synchronizing codeword is at most $\frac{|\mathcal{R}|}{K-|\mathcal{R}|}$ additional bits per encoded bit. Note that suboptimality of the code itself is not counted here.

Such an algorithm does not exploit the tendency of Huffman codes to resynchronize spontaneously. If K is much larger than the average synchronization delay, it is unlikely that the synchronization delay exceeds K in many cases. This means that most of the inserted markers can be removed.

Let us call an insertion of a marker at a position y (y is a codeword boundary) *necessary* if there is a position p in $\mathcal{E}$ such that D_p does not resynchronize before position y, with $y - p > K$, and there are no necessary markers between the bits p and y. In order to see where the markers are necessary, the synchronization delay for every decoder D_p has to be examined. We already know from Section 3.5 two algorithms that track all possible decoders of a Huffman-encoded message and, after each codeword, they are able to tell the largest synchronization delay (so far) of a decoder that is not synchronized at the moment.

The algorithm for inserting only necessary markers is called UNKNOWNSTARTPOSITION and is presented as Algorithm 3.10.

Algorithm 3.10: The encoder for the method UNKNOWNSTARTPOSITION.

Input: K — the synchronization delay parameter ($K > \max(h, \mathcal{R})$), $\mathcal{M}$ — source message, $\mathcal{R}$ — the resynchronization marker

Output: Encoded Message with synchronization markers

```
1  p ← 0;
2  foreach letter l in M do
3      output c(l);
4      p ← p + |c(l)|;
5      process l with DECODERTRACKING;
6      p_min ← UNSYNCHRONIZED(p);          ◁ computed by DECODERTRAKING
7      if p − p_min > K then
8          output R;
9          p ← p + |R|;
10         process R with DECODERTRACKING;
```

Example 3.34: Let us consider the following message encoded with the code C_2 of Figure 3.1.

$$\mathcal{E} = C_2(\mathcal{M}) = \underbrace{0\,1\,0}_{d}{}^{3}\,\underbrace{1\,1}_{c}{}^{5}\,\underbrace{1\,0}_{b}{}^{7}\,\underbrace{0\,0}_{a}{}^{9}\,\underbrace{0\,0}_{a}{}^{11}\,\underbrace{1\,0}_{b}{}^{13}\,\underbrace{0\,1\,0}_{d}{}^{16}. \tag{3.14}$$

The positions of codeword boundaries are numbered. This message will be encoded with the method UNKNOWNSTARTPOSITION for $K = 4$. The code C_2 contains two synchronizing codewords, $c(d) = 010$ and $c(e) = 011$ (see Example 2.46). The latter will be used as the resynchronization marker, as it does not appear in the message $\mathcal{M}$.

The first decoder with the synchronization delay exceeding $K = 4$ is decoder D_4. It is unsynchronized at position 9, so the marker, 011, is inserted there. The beginning of the message is:

$$\mathcal{E}_{(1)} = \underbrace{0\,1\,0}_{d}\underbrace{1\,1}_{c}\underbrace{1\,0}_{b}\underbrace{0\,0}_{a}\underbrace{0\,1\,1}_{e}. \tag{3.15}$$

Now, all decoders that have started before the synchronizing codeword 011 are synchronized. In particular, decoder D_4 is synchronized there. Its synchronization delay is $5 + 3 = 8$. The rest of the codewords are encoded normally, because no synchronization delay of any decoder at a codeword boundary exceeds 4. The result message is the following.

$$\mathcal{E}_{(1)} = \underbrace{0\,1\,0}_{d}\underbrace{1\,1}_{c}\underbrace{1\,0}_{b}\underbrace{0\,0}_{a}\underbrace{0\,1\,1}_{e}\underbrace{0\,0}_{a}\underbrace{1\,0}_{b}\underbrace{0\,1\,0}_{d}. \tag{3.16}$$

On decoding, if the codeword 011 is encountered, it is just ignored. In this way the original message is recovered.

Theorem 3.35. *The method* UNKNOWNSTARTPOSITION *enforces resynchronization for any decoder after at most* $L = K + h + |\mathcal{R}|$ *bits. The redundancy introduced by the insertions of* $\mathcal{R}$ *is at most* $\frac{|\mathcal{R}|}{K-|\mathcal{R}|}$ *additional bits per encoded bit, but the exact value depends on the synchronization properties of the code and on the source message* $\mathcal{M}$.

Example 3.36: The largest synchronization delay of any decoder in the message from Example 3.34 with inserted markers is 8. This is the decoder D_4. The bound on the synchronization delay from Theorem 3.35 is $L = 4 + 2 + 3 = 9$.

The redundancy of inserting the marker in Example 3.34 is $\frac{3}{16}$. Note that it is much smaller than the bound of Theorem 3.35, $\frac{3}{4-1}$, because most of the decoders resynchronize before their synchronization delay reaches 4.

Note that the redundancy introduced by having a dedicated codeword for the resynchronization marker is not counted in Theorem 3.35. It will be shown that this redundancy can be eliminated.

Example 3.37: Character e does not appear in the message from Example 3.34. If this character is eliminated from the code, the codeword for the letter d may be shortened to 01. The message (3.14) encoded with the new code is:

$$\mathcal{E} = \underbrace{0\,1}_{d}\underbrace{1\,1}_{c}\underbrace{1\,0}_{b}\underbrace{0\,0}_{a}\underbrace{0\,0}_{a}\underbrace{1\,0}_{b}\underbrace{0\,1}_{d}. \tag{3.17}$$

The length of the message is reduced by two bits.

Now, a new scheme will be shown, in which the encoder uses any synchronizing string as the resynchronization marker $\mathcal{R}$. The synchronizing string may also appear in the encoded message as encoding of source symbols. The encoding algorithm is exactly the same as previously (Algorithm 3.10), but this time $\mathcal{R}$ is any synchronizing string. The decoding algorithm has to decide whether an encountered marker $\mathcal{R}$ should be ignored or should be decoded as a sequence of letters. It will be shown that this is possible.

The key point of the method is that the decoding algorithm can mimic exactly the operations of the encoder, including the computations of synchronization delays (Algorithm 3.5 or 3.9). The decoding algorithm thus knows when to expect an inserted marker in the encoded message and is able to ignore it. At start, the decoder does not know the synchronization delays of the decoders that started before. It will be shown, however, that after a limited number of bits the decoder has all the necessary information. The decoding algorithm is presented in Algorithm 3.11.

Example 3.38: Let us consider the message from Example 3.34 with $c(e) = 011$ appended at the end:

$$\mathcal{E} = \underbrace{0\,1\,0}_{d}\underbrace{1\,1}_{c}\underbrace{1\,0}_{b}\underbrace{0\,0}_{a}\underbrace{0\,0}_{a}\underbrace{1\,0}_{b}\underbrace{0\,1\,0}_{d}\underbrace{0\,1\,1}_{e}. \tag{3.18}$$

Now every codeword appears at least once in the encoded message. Let the marker $\mathcal{R}$ be again $c(e) = 011$. The message encoded with the method UNKNOWNSTARTPOSITION is:

$$\mathcal{E} = \underbrace{0\,1\,0}_{d}\underbrace{1\,1}_{c}\underbrace{1\,0}_{b}\underbrace{0\,0}_{a}\underbrace{0\,1\,1}_{e}\underbrace{0\,0}_{a}\underbrace{1\,0}_{b}\underbrace{0\,1\,0}_{d}\underbrace{0\,1\,1}_{e} \tag{3.19}$$

Algorithm 3.11: Decoding algorithm for the method UNKNOWNSTARTPOSITION.

Input: K – synchronization delay parameter, $\mathcal{E}$ – message encoded with the method UNKNOWNSTARTPOSITION, L — the letters $c^{-1}(\mathcal{R})$

```
1  p ← 0;
2  while there are bits on the input do
3      decode the next letter l;
4      output l;
5      p ← p + |c(l)|;
6      process l with DECODERTRACKING;
7      p_min ← UNSYNCHRONIZED(p);
8      if p − p_min > K then
9          for i = 0 to |L| − 1 do
10             decode the next letter l;
11             if L[i] ≠ l then UNREAD(l) and break;
12             process l with DECODERTRACKING;
```

Note that the letter e appears in the message twice: once as the resynchronization marker, and this occurrence has to be removed, and once as a part of the source message, and this occurrence should be kept.

Theorem 3.39. *Let $\mathcal{R}$ be any synchronizing string for a Huffman code C. Let $\mathcal{E}$ be a message encoded with C using Algorithm 3.10 with $\mathcal{R}$ as a resynchronization marker and the parameter $K > h + 3|\mathcal{R}|$. Algorithm 3.11 used for decoding $\mathcal{E}[p..)$ will be decoding correct data after having processed at most $L = K + h + 3|\mathcal{R}|$ bits.*

Proof. Recall that a decoder just transforms the bits of $\mathcal{E}$ into codewords and the decoding algorithm, in addition, skips the inserted markers. Decoder D_p will be considered. First, it will be proven that there is a position p' on a codeword boundary in $\mathcal{E}$ such that:

- $p' - p \leq K + h + |\mathcal{R}|$,
- decoder D_p is synchronized at p',
- the content of the set S of the decoding algorithm is the same as the content of S of the encoder at the same position p'.

Let us consider a position p_c in $\mathcal{E}$ that is the first codeword boundary farther than K bits from p. It must be $p_c \leq p + K + h$. We are considering the state of both the encoding and the decoding algorithm at p_c. There are three cases.

1. There is a decoder in the encoder's set S that has been unsynchronized for more than K bits and the encoder will now start inserting $\mathcal{R}$.
2. There are no decoders in the encoder's set S that have been unsynchronized for more than K bits and the encoder is not inserting $\mathcal{R}$ now.

3. The coder is in the middle of inserting $\mathcal{R}$ for a decoder that had been unsynchronized for more than K bits before position p_c.

In the first and third cases, decoder D_p will be synchronized after receiving $\mathcal{R}$, as $\mathcal{R}$ resynchronizes any decoder. This happens not later than at position $p' = p_c + |\mathcal{R}| \leq p + K + h + |\mathcal{R}|$. The sets S of both the coder and the decoding algorithm at p' contain only the decoders that started inside the inserted marker, so they are equal.

In the second case, the set S of the encoder at p_c cannot contain any decoder that started before position $p_c - K > p$. It follows that D_p is not there and thus D_p must already in synchronization. The set S of the decoding algorithm is equal to the one of the encoder. In this case $p' = p_c$.

To complete the proof it is necessary to show that the decoder is able to distinguish letters of the inserted markers from letters of $\mathcal{M}$. This is not trivial because $\mathcal{R}$ may also appear as a part of the message $\mathcal{M}$ and, in this case, must not be removed during decoding.

If the decoding algorithm is at position p' or farther in $\mathcal{E}$ and determines correctly that one codeword corresponds to a letter of $\mathcal{M}$, the rest of the data is decoded correctly. We consider the three cases again. In cases 1 and 3, after the encoder finishes inserting a marker it will not insert another one in the next $K - |\mathcal{R}| > |\mathcal{R}|$ bits. At p' the decoding algorithm may be in the middle of skipping a marker in the loop in line 9. After less than $|\mathcal{R}|$ bits it finishes and decodes correctly at least one letter of $\mathcal{M}$. In these cases the decoder will be decoding correct data after having processed at most $K + h + 2|\mathcal{R}|$ letters, which is better by $|\mathcal{R}|$ than needed.

In case 2, the encoding algorithm at $p' = p_c$ encodes a letter of $\mathcal{M}$, but the decoding algorithm may be in the middle of skipping a marker (the loop in line 9). When it finishes, either by skipping all the letters of $\mathcal{R}$ or by executing the break instruction in line 11, it has to decode at least least one letter of $\mathcal{M}$ (line 3). At this position the encoder may either encode a normal letter or a letter of an inserted marker. In the former case the theorem is proved as the decoder decodes correctly a letter of $\mathcal{M}$. In the latter case the situation is equivalent to either case 1 or 3, but the decoding algorithm is at most $|\mathcal{R}|$ bits farther now. The bound for these two cases was better by $|\mathcal{R}|$ bits then needed so the theorem is proved. □

Example 3.40: Let us consider the decoding of the message (3.19) from Example 3.38 (note that the value of $K = 4$ does not meet the requirements of Theorem 3.39, but the decoding will still work well in this case). Let us start with decoder D_0. After decoding the first codeword, the value of the largest synchronization delay is equal 1. Then, after the second and third codeword, it is equal 1 and 3, respectively. After the next codeword, 00, the synchronization delay is 5, so the decoder has to skip the inserted resynchronization marker. The next codeword, 011, is indeed a resynchronization marker and it is ignored. The decoding continues. Before the last letter, e, the largest synchronization delay of a decoder is 1, so the codeword is decoded as a source letter.

Now let us consider decoder D_1. It resynchronizes after reading the first two bits and from then on it operates in the same way as decoder D_0. Decoder

D_2 is in synchronization immediately. Decoder D_3 resynchronizes at position 5 and then it operates as D_0.

The situation is different for decoder D_4. This is the decoder that is unsynchronized for too long and is synchronized with a synchronizing codeword. First, it decodes incorrectly three codewords, 11, 00 and 00. Then it notices that decoder D_{4+1}, that is the decoder that starts at position 1 with respect to the start position of D_4, has been unsynchronized with respect to D_1 for 5 bits. Decoder D_4 is now trying to skip the resynchronization marker. When it reads the next codeword, 11, it sees that it is not a part of the marker and decodes it as a normal symbol, c. Now, it is in synchronization and operates as D_0.

The operation of decoder D_5 is also differs from the operation of D_0. Even though D_5 is synchronized at start, at position 9 the decoder with the largest synchronization delay is D_{5+1}, which has been unsynchronized for 3 bits. So D_5 does not expect a synchronizing codeword and decodes the next bits, 011, as a source letter, incorrectly. After that, its information about unsynchronized decoders is the same as what D_0 knows, and both decoders work in the same way. Note that resynchronization in this case occurred after 7 bits, which is less than the bound of Theorem 3.39: $4 + 3 + 9 = 16$.

There is a way to reduce slightly the redundancy inserted by the encoder. In the set S there are at most h decoders that have to be synchronized (Lemma 3.23). Instead of inserting a synchronizing string for the code, it is enough to insert a merging string for the at most h decoders that are currently in S. Such a string may be shorter than a synchronizing string.

The contents of the set S is fully determined by the lowest node q_l (see Section 3.5.2), so there are at most $N - 1$ possible values for S. The merging string for each of these $N - 1$ sets can be precomputed. Nevertheless, as the following lemma states, the gain of using this optimization is not substantial.

Lemma 3.41. *Let W be a set of states that is a possible contents of the set S for Algorithm 3.10, after processing at least h bits. Let s be the shortest synchronizing string for the Huffman automaton $\mathcal{T}$ used for encoding, and let s_W be the shortest merging string for W. Then*

$$|s_W| \leq |s| \leq |s_W| + h. \tag{3.20}$$

Proof. The first inequality follows from the definitions of a synchronizing string and a merging string. For the second inequality, let w be the last h bits processed by Algorithm 3.10 before the contents of S became W. From $\delta^*(Q, w) = W$ follows that ws_W is a synchronizing string. □

Now, it will be shown that any bit error propagates for a limited number of bits. Let us consider a decoder in synchronization. A bit error may desynchronize the decoder and may also influence the contents of the set S. Let D_e be the erroneous decoder. Decoder D_e is always in the same state as decoder D_{p_1}, that started when D_e completed the codeword that contained the error. Position p_1 is at most $h - 1$

bits after the error. If decoder D_{p_1} does not resynchronize in about K bits, the encoder inserts a synchronizing string. Such string resynchronizes the decoder no later than at position $p_1 + K + h + |\mathcal{R}|$. By repeating the argumentation from the proof of Theorem 3.39, at this position decoder D_e will have the same content of the set S as the encoder. Then, again by the arguments from the proof of Theorem 3.39, after at most $2|\mathcal{R}|$ more bits decoder D_e will decode correct symbols. This may be summarized as follows.

Theorem 3.42. *Let $\mathcal{R}$ be the synchronizing string for the method* unknownStartPosition *with parameter $K > h + 3|\mathcal{R}|$. Any bit error in a message encoded with this method influences the decoding for at most $L = K + 2h + 3|\mathcal{R}|$ bits.*

Note that if an error occurs during the resynchronization process, it may also propagate for the same number of bits.

3.7 No-subword resynchronization marker

The methods knownStartPosition and unknownStartPosition, described in Sections 3.4 and 3.6, assure that any decoder resynchronizes after a limited number of bits. Nevertheless, a decoder that loses synchronization after a bit error may decode a wrong number of symbols. In many applications placing following symbols at wrong positions still is unacceptable.

In this section we are interested in *strong synchronization* (see Section 1.2.6), which allows for placing decoded symbols at correct positions. It will be provided by insertions of some resynchronization marker (RM), called a *no-subword resynchronization marker* (no-subword RM) for the method used for its construction. The marker is constructed in such a way that the decoder always recognizes any of such insertions. In contrast, in the methods knownStartPosition and unknownStartPosition the resynchronization string was received implicitly by the decoder and an unsynchronized decoder was not able to find out which bits corresponded to the string that resynchronized it.

The goal that the decoder is always able to recognize any insertion of the resynchronization marker is the same as in case of the extended synchronizing codeword (ESC) [40, 55] (see also Section 1.2.6). Both methods solve exactly the same problem but using a different approach. This is why the results will later be compared with ESC.

Strong synchronization can be achieved by providing some positional information together with each no-subword RM. The form of the additional information was discussed in depth in other work (see Section 1.2.6 for references). Nevertheless, for completeness, two examples are outlined below.

The simplest method to assure strong synchronization of the decoded data is to insert the no-subword RM at some intervals, followed by the position of the current symbol. A decoder can always recognize each inserted marker, so it knows where to find the positional information that was encoded. Then, it is able place the following symbols correctly. Note that this method also allows for the decoding of a message

starting from an arbitrary bit. Indeed, the decoder skips bits until a marker is found, then it reads the current symbol's position and the following symbols are placed correctly.

Another approach to provide the decoder with symbols' positions is to place the marker in $\mathcal{E}$ after each K-th codeword. Then, assuming for simplicity that there is no bit error in any marker, the decoder can count the number of occurrences of the marker. It then knows the number of symbols encoded so far and this gives the position of the next symbol. This method does not give the correct symbol's position to a decoder that starts at an arbitrary bit in the encoded message.

Let us discuss the design goals from Section 3.4 too see how they apply to the new method. The first goal was to keep the code optimal. This will indeed be the case, but the resynchronization marker itself will have to be transmitted explicitly to the decoder. The length of the marker grows only logarithmically with the length of the encoded message. On the other hand, any suboptimality of the code, like in case of ESC, introduces redundancy that grows linearly with message's length.

The second goal was to make use of spontaneous synchronization of Huffman codes. This time the goal cannot be achieved because strong synchronization of Huffman codes is not recovered spontaneously.

Let us consider a source message $\mathcal{M}$ and its encoding $\mathcal{E}$. The no-subword resynchronization marker will depend on $\mathcal{E}$ and will be constructed from a word w_0 that is not a subword of $\mathcal{E}$.

Definition 3.43. A *no-subword* for an encoded message $\mathcal{E}$ is a word that is not a subword of $\mathcal{E}$.

Lemma 3.44. *There exists a no-subword w_0 for $\mathcal{E}$ such that $|w_0| \leq \lceil \log |\mathcal{E}| \rceil$.*

Proof. Let $|\mathcal{E}| = L$. The message, $\mathcal{E}$, has less than L subwords of length $M = \lceil \log L \rceil$. The number of different bit sequences of length M is $2^M = 2^{\lceil \log L \rceil} \geq L$. Thus, there is a string w_0 of length M that does not appear in $\mathcal{E}$. □

It is tempting to use any no-subword as a resynchronization marker, but it can be used only under certain conditions. If a no-subword w_0 has a proper prefix $w_0' \neq \epsilon$ that is also a suffix of w_0, i.e. $w_0 = w_0' w_s$ and $w_0 = w_p w_0'$, it cannot be used as a resynchronization marker. In such case, spurious occurrences of w_0 may appear in $\mathcal{E}$. If w_0, as the marker, was inserted just after an occurrence of w_p in the message, the message would contain a subword $w_p w_0' w_s$, which can be decoded in two ways, either correctly as $w_p(w_0' w_s) = w_p w_0$ or incorrectly as $(w_p w_0')w_s = w_0 w_s$.

Example 3.45: Let us consider a no-subword 01001 for the encoded message 11010110. Let the no-subword be inserted after the fifth bit of the message. The result will be the following:

$$\mathcal{E} = \underbrace{1\,1\,0\,1\,0}_{\text{msg}}\ \underbrace{0\,1\,0\,0\,1}_{\text{marker}}\ \underbrace{1\,1\,0}_{\text{msg}}. \tag{3.21}$$

The decoder will detect an insertion of the marker at position 2:

$$\mathcal{E} = \underbrace{1\,1}_{\text{msg}}\ \underbrace{0\,1\,0\,0\,1}_{\text{marker}}\ \underbrace{0\,0\,1\,1\,1\,0}_{\text{msg}}, \tag{3.22}$$

which is incorrect.

The following definition describes the construction of a proper marker.

Definition 3.46. A *no-subword resynchronization marker* for an encoded message $\mathcal{E}$ is a word $\mathcal{R} = w_0 v$ such that w_0 is a no-subword for $\mathcal{E}$, and v is such a string that no proper prefix p of $\mathcal{R}$ of length $|p| > |v|$ is a suffix of $\mathcal{R}$.

The construction of a no-subword resynchronization marker is based on the following theorem, which is proved in the Appendix.

Theorem 3.47. *For any binary word w there is a binary word v, $|v| \leq \lfloor \log\log|w| \rfloor + 1$, such that if $wv = xp$, where x is any word and p is a prefix of wv, then either $x = \epsilon$ or $|x| \geq |w|$.*

Example 3.48: For the marker 01001 from Example 3.45 the extension is 1. Indeed, in the word 010011 no suffix is a prefix.

The extension for the word 01110 is 0, even though the suffix of the extended word 011100 of length 1 is a prefix of 011100. This is is correct, because the suffix is not longer than the extension.

From Lemma 3.44 and Theorem 3.47 follows directly the bound on the length of the shortest no-subword resynchronization marker.

Corollary 3.49. *For any encoded message $\mathcal{E}$ of length L there is a no-subword resynchronization marker of length at most $\lceil \log L \rceil + \lfloor \log\log\lceil \log L \rceil \rfloor + 1$.*

Assume that a no-subword resynchronization marker $\mathcal{R} = w_0 v$, where w_0 is a no-subword, is known to both the coder and the decoder. Let us consider the message $\mathcal{E}'$ that is $\mathcal{E}$ with inserted markers $\mathcal{R}$. We assume that the number of bits between each insertion of $\mathcal{R}$ is at least $|\mathcal{R}|$. It will be shown that any insertion of $\mathcal{R}$ can be detected correctly by the decoder.

Let us consider a decoder starting at an arbitrary bit of $\mathcal{E}'$. We assume that the decoder does not start inside of an insertion of $\mathcal{R}$ (such a case would make the recognition of this particular insertion of the marker impossible, but the decoder would synchronize at the next one). Let us study the first occurrence of $\mathcal{R}$ in the fragment of $\mathcal{E}$' processed by the decoder. Let this occurrence of $\mathcal{R}$ be denoted by R. It is enough to show that R is an inserted no-subword resynchronization marker.

This is indeed the case. No inserted marker can appear before R, because this is the first occurrence of $\mathcal{R}$ in $\mathcal{E}'$. An inserted marker has to have at least one common bit with R, because otherwise $\mathcal{R}$ would appear in $\mathcal{E}$, which would contradict the definition of a no-subword resynchronization marker. R may be formed of a part of $\mathcal{E}$, the original encoded message, and of a prefix of another $\mathcal{R}$ inserted, i.e. $R = xp$, where p is a prefix of $\mathcal{R}$ and x appears in $\mathcal{E}$. But this contradicts Definition 3.46. On one hand $|x| < |w_0|$ since w_0 cannot appear in $\mathcal{E}$, on the other hand $|x| \geq |w_0|$ because $|p|$ is a proper prefix of $\mathcal{R}$. This proves that R is an insertion of $\mathcal{R}$.

Let us now consider a decoder that starts inside an inserted $\mathcal{R}$. It may happen that a suffix, v', of the extension v forms $\mathcal{R}$ with the bits of $\mathcal{E}$ that follow it (note that

v' is allowed to be a prefix of $\mathcal{R}$). In such a case, this particular marker is recognized incorrectly, but the decoder will recognize correctly the next insertion of $\mathcal{R}$.

A decoder can detect the inserted markers by executing any string-matching algorithm, e.g. KMP, see [15], to find the first occurrence of $\mathcal{R}$ in $\mathcal{E}'$. Then, the insertion is removed and the string matching algorithm is run again on the remaining fragment of $\mathcal{E}$'. This process may be done at the time of decoding $\mathcal{E}$'.

Note that the no-subword resynchronization marker may be used instead of a synchronizing string in the method UNKNOWNSTARTPOSITION, Algorithm 3.10. An important advantage in this case is that the decoder does not have to execute the lengthy algorithm for decoder tracking, as it is always able to detect any insertions of the resynchronization marker using a string matching algorithm.

A disadvantage of the method is the process of finding a short no-subword. An algorithm for this problem may be the following. First, create a bit array indexed with all possible strings of a fixed length $L = \lceil \log |\mathcal{E}| \rceil$. Then, read the message $\mathcal{E}$ and mark in the array the occurrences of any string of length L by setting the corresponding bit. At the end, return some index with an unset bit. Lemma 3.44 guarantees that at least one such index exists.

In real applications there may not be enough memory for all the $2^L \geq |\mathcal{E}|$ strings of length L. In spite of that, an iterative approach may be used to find a no-subword. It involves the following steps. First, count the number of zeros, n_0, and ones, n_1, in the binary message $\mathcal{E}$. The total number of bits is $|\mathcal{E}|$, so one of the numbers is less or equal $\lfloor |\mathcal{E}|/2 \rfloor$. Let us assume that it is n_0. In the second pass, count the number of subwords 00, n_{00}, and 01, n_{01}. The sum of these two numbers is at most n_0 (at most, because the last bit does not have a pair), so at least one of them is less or equal $\lfloor n_0/2 \rfloor \leq \lfloor |\mathcal{E}|/4 \rfloor$. This process can be continued to find a no-subword in at most $\lceil \log |\mathcal{E}| \rceil$ steps.

The process above can be improved by counting longer strings of bits. For instance, in the first pass the number of occurrences of each 8-bit binary string can be counted. Then, the string w_1, with least occurrences is chosen and the 16-bit binary strings starting with w_1 are counted, and so on.

In typical applications, 16-bit strings can easily be counted in one pass (this requires 65536 words of memory, which is 256kB with 32-bit integers). In this case, a no-subword may be found with two passes in messages of size not exceeding 4 billion bits (512 MB).

It may be expected that there is a no-subword that is shorter than the bound from Lemma 3.44 (see also Section 3.8.5 for numerical tests). If the number of bits is set to a larger value, for instance 20, it may be expected that the no-subword is found in the first pass, even if the upper bound on the minimal length is, for instance, 24. Only if the no-subword is not found in the first pass, the second pass is applied. Unfortunately, these methods have the drawback that the no-subword they give is not necessarily the shortest one. However, the increase in size is just a few bits.

The necessity to preprocess the data to construct a no-subword may be regarded as unrealistic for real applications. It should be pointed out, however, that preprocessing the input is also present when frequencies of letters are counted for creating a Huffman code. Thus, more passes through the data may often be tolerable.

File name	File description	Type	Size [MB]
dickens	Collected works of Charles Dickens	English text	9.72
mozilla	Tarred execs of Mozilla 1.0 (Tru64 UNIX ed.)	exe	48.85
mr	Medical magnetic resonance image	picture	9.51
nci	Chemical database of structures	database	5.87
ooffice	A dll from Open Office.org 1.01	exe	5.87
osdb	Sample database in MySQL format	database	9.62
reymont	Text of the book Chłopi by Władysław Reymont	Polish pdf	6.32
samba	Tarred source code of Samba 2-2.3	src	20.61
sao	The SAO star catalog	bin data	6.92
webster	The 1913 Webster Unabridged Dictionary	html	39.54
x-ray	X-ray medical picture	picture	8.08
xml	Collected XML files	html	5.10

Table 3.7: Files from the Silesia corpus.

3.8 Results of numerical tests

This section contains the result of numerical tests for the algorithms presented in former sections.

3.8.1 Test files

The algorithms introduced in this chapter were tested on files from the Silesia Corpus [17]. Silesia Corpus is a set of 12 files that covers typical data types used nowadays. The files are between 6 MB and 51 MB long. Their description is presented in Table 3.7.

The files were compressed using a few different settings. Firstly, they were compressed using bytes of the files as source letters. Secondly, the files were first compressed with the LZW algorithm [71] with dictionary of size 4096 and then the output (12-bit words) was compressed with a Huffman code. The dictionary was fixed after it filled up. The Huffman codes' sizes were, in these cases, close to 4096. The latter tests are important because Huffman coding is often used in cascaded compression with other algorithms, for instance with dictionary methods. It is also important to see how the algorithms behave for small and large codes.

Two Huffman codes were used: canonical Huffman codes, in which longer codewords always lexicographically precede shorter ones, and codes constructed using the *fixed order method* of Zhou and Zhang [74]. The former codes are used for faster decompression and smaller overhead for code's transmission. The latter, hereinafter called *fixed-order* codes, exhibit very low average synchronization delay.

General properties of Huffman codes for each of the test files are presented in Table 3.8. Note that these values are the same for any Huffman code for a given file. Table 3.9 contains the numbers of codewords of length 1, 2, ... (the length vectors) in Huffman codes for these files. The first value of each list in the right column is

File name	Without LZW				With LZW			
	$\|\mathcal{M}\|$ [MB]	N	h	$\|\mathcal{E}\|$ [MB]	$\|\mathcal{M}\|$ [MB]	N	h	$\|\mathcal{E}\|$ [MB]
dickens	9.72	100	23	5.56	5.39	3773	22	4.55
mozilla	48.85	256	11	38.12	49.36	3940	25	35.11
mr	9.51	256	11	4.41	6.11	2978	22	3.89
nci	32.00	62	20	9.75	5.25	3773	22	4.34
ooffice	5.87	256	12	4.89	5.47	3704	22	4.26
osdb	9.62	256	12	7.76	8.05	3985	22	6.78
reymont	6.32	256	16	3.84	2.51	3798	21	2.16
samba	20.61	256	12	15.78	18.71	4013	24	13.60
sao	6.92	256	9	6.53	7.70	3964	22	5.71
webster	39.54	98	21	24.73	33.15	3738	24	21.44
x-ray	8.08	256	10	6.70	9.27	3931	23	6.67
xml	5.10	104	22	3.54	4.85	2978	22	2.98

Table 3.8: Huffman codes for files from the Silesia corpus.

the number of codewords of length 1, the second — of length 2, and so on. This information is also independent of a particular Huffman code

Although some of the length vectors presented in Table 3.9 are gapless, that is there is no zero after the first nonzero entry, there are no gapless length vectors with 2 as the minimal codeword length. These two conditions, that is being gapless and having the shortest codeword of length 2, are prerequisites for the method of Ferguson and Rabinowitz [22] for the construction of codes with a synchronizing codeword (see also Section 1.2.4). It follows that their method cannot be used for any of our test files, so, in general, their method is rather of theoretical interest only.

The greatest common divisor of the codewords' lengths is in each case equal 1, so by the main theorem of [65], a code with a synchronizing string can always be constructed for these length vectors.

3.8.2 Estimating the synchronization delay

Figure 3.2 presents a histogram of the synchronization delay's estimates found by Algorithm 3.1. The tests were performed on the file *mozilla*. The file was compressed with the four compression methods described in Section 3.8.1.

Algorithm 3.1 was executed for each start position in the encoded message of the form $(100 \cdot i)$, $i = 0, 1, \ldots, \lfloor |\mathcal{E}|/i \rfloor$. The results were grouped into bins with ranges shown on the x-axis in Figure 3.2. The fixed-order codes have good synchronization properties so the number of bits before resynchronization found by Algorithm 3.1 is lower than for canonical Huffman codes.

Although the delay found by Algorithm 3.1 is in almost all cases small, of the order of tens of bits, the maximum synchronization delay over all start positions (not only the ones of the form $100 \cdot i$) is large and equals 16460, 3112, 22921 and 6351 bits for the four cases of Figure 3.2, in the same order as the legend. Note that these

File name	Length Vector
dickens	[0, 0, 2, 6, 4, 12, 4, 1, 5, 10, 11, 7, 2, 4, 4, 5, 3, 2, 5, 4, 1, 4, 4]
mozilla	[0, 1, 0, 0, 1, 7, 29, 42, 58, 98, 20]
mr	[1, 0, 1, 0, 0, 10, 4, 5, 10, 79, 146]
nci	[1, 1, 0, 0, 4, 4, 5, 1, 3, 7, 6, 14, 2, 2, 3, 1, 0, 1, 5, 2]
ooffice	[0, 0, 1, 1, 4, 11, 17, 45, 62, 63, 48, 4]
osdb	[0, 0, 0, 1, 7, 9, 58, 3, 5, 63, 62, 48]
reymont	[0, 0, 2, 4, 6, 10, 13, 9, 2, 5, 3, 15, 4, 1, 150, 32]
samba	[0, 0, 1, 1, 9, 11, 26, 21, 10, 11, 14, 152]
sao	[0, 0, 0, 0, 3, 9, 23, 79, 142]
webster	[0, 0, 1, 5, 11, 8, 4, 5, 11, 16, 5, 8, 9, 5, 1, 1, 1, 0, 2, 3, 2]
x-ray	[0, 0, 1, 2, 9, 0, 1, 1, 226, 16]
xml	[0, 0, 0, 4, 10, 15, 15, 12, 15, 6, 5, 5, 0, 2, 1, 4, 3, 1, 1, 0, 3, 2]
dickens.lzw	[0, 0, 0, 0, 0, 1, 6, 38, 104, 207, 358, 446, 472, 378, 338, 341, 292, 337, 112, 92, 69, 182]
mozilla.lzw	[0, 0, 0, 0, 0, 12, 24, 45, 109, 128, 45, 117, 174, 288, 362, 353, 397, 398, 296, 237, 244, 219, 172, 88, 232]
mr.lzw	[0, 0, 2, 0, 0, 4, 5, 26, 139, 107, 192, 163, 134, 146, 159, 207, 336, 237, 120, 152, 127, 722]
nci.lzw	[0, 0, 0, 0, 0, 0, 11, 26, 150, 199, 315, 377, 414, 316, 199, 113, 102, 94, 86, 92, 129, 1150]
ooffice.lzw	[0, 0, 0, 0, 0, 3, 15, 65, 111, 134, 189, 246, 291, 441, 312, 354, 328, 348, 254, 245, 140, 228]
osdb.lzw	[0, 0, 0, 0, 0, 0, 20, 50, 21, 95, 358, 970, 509, 287, 672, 85, 53, 123, 193, 192, 259, 98]
reymont.lzw	[0, 0, 0, 0, 0, 0, 3, 23, 108, 288, 406, 386, 557, 354, 210, 230, 166, 159, 111, 143, 654]
samba.lzw	[0, 0, 0, 1, 0, 13, 17, 36, 46, 79, 301, 225, 334, 399, 473, 460, 440, 355, 254, 181, 130, 76, 47, 146]
sao.lzw	[0, 0, 0, 0, 2, 1, 8, 78, 196, 45, 88, 124, 110, 194, 192, 548, 1869, 183, 59, 61, 86, 120]
webster.lzw	[0, 0, 2, 0, 2, 5, 15, 23, 30, 72, 191, 308, 469, 449, 387, 329, 246, 222, 213, 221, 254, 60, 64, 176]
x-ray.lzw	[0, 0, 0, 2, 6, 2, 1, 4, 121, 146, 90, 303, 743, 427, 382, 285, 210, 199, 221, 304, 208, 123, 154]
xml.lzw	[0, 0, 0, 1, 3, 12, 33, 37, 62, 57, 66, 69, 93, 103, 148, 180, 183, 193, 193, 174, 196, 1736]

Table 3.9: The length vectors of Huffman codes for the files from the Silesia corpus.

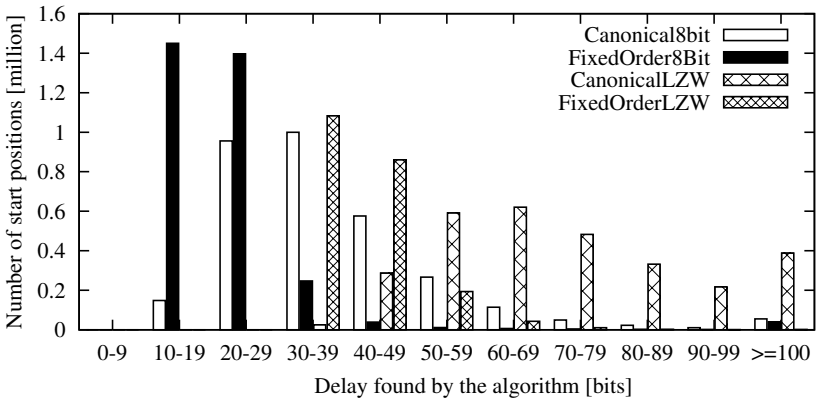


Figure 3.2: Histogram of the number of bits before resynchronization found by Algorithm 3.1 for the file *mozilla* with four different codes.

Redundancy for:			Canonical HC			Fixed order HC [74]		
File	compr. size	h	$K = 100$	200	500	$K = 100$	200	500
dickens	5.6MB	23	32	0	0	0	0	0
mozilla	38.1MB	11	63506	4359	431	552	80	11
samba	15.8MB	12	16758	1349	55	1107	128	3
xml	3.5MB	22	395	5	0	0	0	0
dickens.lzw	4.54MB	22	39571	4397	21	625	4	0
mozilla.lzw	35.1MB	25	189982	20824	1180	2114	269	62
samba.lzw	13.6MB	24	101483	13315	484	672	106	10
xml.lzw	3.0MB	22	10340	884	9	8	0	0

Table 3.10: The number bits inserted by Algorithm 3.3 (knownStartPosition) for selected files from the Silesia Corpus.

results are real delays and not the delay's estimates found by Algorithm 3.1, which are, in most cases, longer.

3.8.3 Known start position

Table 3.10 presents the redundancy, in bits, introduced by Algorithm 3.3 for K equal 100, 200 and 500. The second column gives the size in mega bytes of each file after compression. The third column is the length, in bits, of the longest codeword. Table 3.10 shows that the inserted redundancy is negligible even if the parameter K is only 10-20 times the length of the longest codeword. The redundancy is much smaller if the code itself synchronizes well, as in the case of the fixed-order codes.

The time overhead of Algorithm 3.3 over normal Huffman coding was in case of direct Huffman coding (without LZW) on average 16% for the sublinear implementation with precomputed synchronization graph, and 60% for the linear implementation with transitions by single bits. The codes contained less than 256 elements, so the quadratic preprocessing time was irrelevant. The codes' sizes for the LZW-

File	size	compr.	h	$\lvert\mathcal{R}\rvert$	max del. [b]	redun. [‰]	A3.5	A3.9
dickens	9.7MB	5.6MB	23	10 / 5	226/ 106	0.0 /0.0	5.4	4.7
mozilla	48.8MB	38.1MB	11	13 / 7	16460/ 3112	.05 /0.02	6.8	4.8
samba	20.6MB	15.8MB	12	13 / 8	14419/ 1107	0.15 /0.05	7.2	5.0
xml	5.1MB	3.5MB	22	9 / 6	331/ 141	0.002/0.0	6.4	4.8
dickens.lzw	5.4MB	4.54MB	22	20 / 8	934/ 306	0.06 /0.003	12.8	6.0
mozilla.lzw	49.3MB	35.1MB	25	21 / 8	6351/ 1433	0.3 /0.06	8.4	4.8
samba.lzw	18.7MB	13.6MB	24	20 / 8	12485/12388	0.4 /0.06	9.3	5.0
xml.lzw	4.8MB	3.0MB	22	17 /10	2773/ 287	0.05 /0.0	9.9	4.6

Table 3.11: Performance of Huffman coding with UNKNOWNSTARTPOSITION method.

precompressed files were around 2^{12} and the preprocessing in $O(N^2)$ turned out to be too slow. The total overhead was in this case on average 300% for the sublinear implementation and 65% for the linear implementation. For long files, however, such as *mozilla*, the overhead for the sublinear implementation was only 60% — the same as in case of the linear implementation.

3.8.4 Unknown start position

The results for the method UNKNOWNSTARTPOSITION are shown in Table 3.11. The first four rows are files compressed directly with a Huffman code, the other rows are files precompressed with LZW. The four first columns of Table 3.11 are the file name, the size before Huffman compression, the size after Huffman compression (the size of the Huffman code's representation was not counted) and the height of the Huffman tree.

Each file was tested with a canonical Huffman codes and with a fixed-order code, so the next three columns contain two values, the result for the former code on the left-hand side and for the latter code on the right-had side. The column $\lvert\mathcal{R}\rvert$ is the length of the shortest synchronizing string for the two codes — this string was used as the resynchronization marker. The next column is the maximum synchronization delay in $\mathcal{E}$ without any marker's insertions. The column *redun.* gives the redundancy (in per mil) introduced by the method UNKNOWNSTARTPOSITION, for the two codes. The parameter K was chosen to be $30|\mathcal{R}|$ and is different for each file and code. Fixed-order codes have better synchronization properties so the redundancy added by the algorithm is lower.

Finally, the last two columns give the ratio of the execution time for UNKNOWNSTARTPOSITION to normal Huffman coding. Tests were done with Algorithm 3.5 (column A3.5) and with Algorithm 3.9 (column A3.9) used for tracking decoders.

Table 3.11 shows that even if the shortest synchronizing string is short and the code has good synchronization capabilities, the synchronization delay may be large. This is possible even for fixed-order codes — in the file *samba.lzw* the largest synchronization delay is 1500 times the length of the shortest synchronizing string.

The results show that with little additional redundancy one can guarantee an

upper bound on the synchronization delay. The method does not introduce any redundancy if decoders of the encoded message synchronize spontaneously, for instance for the file *dickens*. The price for the synchronization is the processing time increased by, on average, 7-8 times in case of Algorithm 3.5 and 5 times in case of Algorithm 3.9. Unlike in case of Algorithm 3.9, the time needed for Algorithm 3.5 depends on the code size and increases with larger codes. This is mainly due to long preprocessing time, which for the lzw files was a fraction of 27%, 5.7%, 14% and 43% of the total time (in the same order as in Table 3.11). In other cases the preprocessing time was negligible.

3.8.5 No-subword resynchronization marker

In the tests of the no-subword resynchronization marker method, the length of the no-subword RM was compared to the length of the extended synchronizing codeword [40, 55]. Tests were performed, as before, on the files from the Silesia Corpus [17] encoded with and without LZW compression. This time only canonical Huffman codes were used for encoding.

The results are presented in Table 3.12. The first column is the file name. The second one is the upper bound on the length of the shortest no-subword, given in Lemma 3.44. This value turned out to be the same for the encodings with both small (no LZW — 8-bit) and large (with LZW — 12-bit) codes. Next, the results for small (8-bit) and large (12-bit) codes are presented. The column h is the length of the longest codeword. If the marker is going to appear rarely in the encoded message, this codeword is the best choice for the base of the extended synchronizing codeword of [40, 55]. The columns [40] and [55] present the lengths of the ESC of the corresponding reference, done by extending the longest codeword. Finally, $|\mathcal{R}|$ is the length of the no-subword RM. In most cases it was enough to extend the no-subword by one bit (see Lemma 3.44 and Theorem 3.47). In two cases no extension was necessary and in six cases two bits were needed.

Table 3.12 shows that, on average, the no-subword RM method gives noticeably shorter markers than ESC of [40] and [55]. Both referenced methods also require that the code has one more codeword than the number of letters in the alphabet. The additional codeword is extended to create an ESC. Such an approach introduces additional redundancy to the code. This kind of redundancy is not present in the no-subword RM method.

One should be aware that the method of [55] allows for using a shorter codeword and not necessarily the longest one, as in Table 3.12, as the base for the ESC. The price for this is increased redundancy in the code. Such approach may be preferable if the insertions of the marker are frequent.

3.9 Applications

The methods presented in this chapter can be used to allow for decoding a Huffman-encoded message from an arbitrary position. This application is important, for in-

File	$\lvert w_0\rvert$	8 bit				12 bit			
		h	[40]	[55]	$\lvert\mathcal{R}\rvert$	h	[40]	[55]	$\lvert\mathcal{R}\rvert$
dickens	24	23	27	26	**14**	22	31	28	**20**
mozilla	26	11	17	16	**18**	25	34	31	**20**
mr	24	11	19	15	**13**	22	32	27	**15**
nci	25	20	24	23	**9**	22	33	29	**16**
ooffice	23	12	18	16	**18**	22	31	28	**20**
osdb	24	12	18	16	**17**	22	32	29	**20**
reymont	23	16	23	20	**10**	21	31	28	**18**
samba	25	12	20	16	**17**	24	33	29	**19**
sao	23	9	17	14	**20**	22	32	27	**21**
webster	26	21	25	24	**13**	24	33	28	**13**
x-ray	24	10	17	15	**15**	23	33	28	**18**
xml	23	22	26	26	**13**	22	33	26	**16**
Average:			20.9	18.9	**14.8**		32.3	28.2	**18.0**

Table 3.12: The length of a no-subword RM for files from the Silesia Corpus.

stance, in information retrieval system, where a large piece of data is compressed with a Huffman code. It allows the decoder to decompress only a fragment of the data.

The method for estimating the synchronization delay (Algorithm 3.1 from Section 3.2) may be used for such decoding of normal Huffman data. The decoder starts some number of bits before the desired fragment. As resynchronization of Huffman codes is quick, in most cases the synchronization will be detected before the beginning of the fragment. If not, the decoder moves some number of bits backwards and restarts the process.

Although this approach to decoding a fragment of Huffman-encoded data works well in most cases, there is no guarantee that the decoder does not have to go back to the beginning of the data. To have such a warranty, a method for guaranteed synchronization, either knownStartPosition or unknownStartPosition, has to be used.

The method knownStartPosition can only be applied if the decoder knows the position where it starts. In most applications of this kind this indeed can be assumed. Otherwise, the method unknownStartPosition can be used. Even though the latter method is more general, the first method has the advantage of being much simpler and faster. Also the introduced redundancy is lower.

The method of estimating the synchronization delay can also be used for direct pattern matching in Huffman-compressed data. To avoid decompression, the pattern can be compressed and then matched to compressed data. Of course, many of the matches are false-positives, because the match may be misaligned with codewords.

To confirm or reject such a match, the method for estimating the synchronization delay can be started a few bits before the match and it can check whether synchronization is acquired before the beginning of the match. If so, it can be answered with 100% accuracy whether or not the match is aligned correctly. Otherwise, the algorithm can restart a number of bits earlier. In most cases the synchronization

will occur in the first iteration, even if only a few tens of bits before the match are analyzed.

A similar approach was suggested by Klein and Shapira [37]. They also moved the decoder a number of bits before the match, in order to synchronize it. However, unlike here, they did not use any method to confirm that such a decoder is synchronized at start of the match and they assumed that it always is. Their method, although in most cases correct, gives some false-positives and false-negatives. On the contrary, the above-mentioned method answers with 100% accuracy.

Even if the number of backward jumps is limited to one, as in [37], this new method is certain in most cases (note that the method of [37] is never certain). In other cases, when the synchronization cannot be confirmed, the answer is still uncertain, but the number of uncertain answers is much lower.

Another application for the methods from this chapter is recovery of compressed data when a prefix is missing (note that the code itself must be available). The decoder can skip some initial bits before it is synchronized. The synchronization can be detected using Algorithm 3.1, if normal Huffman coding was used for encoding, or is guaranteed after a certain number of bits, if the method knownStartPosition or unknownStartPosition was used. In this application it is more likely that the number of missing bits is also unknown and then only the method unknownStartPosition is suitable.

The methods for estimating the synchronization delay can be used together with the methods for guaranteed synchronization. In guaranteed synchronization it is assumed that the decoders resynchronize in most cases before the limit on the synchronization delay. This means that there is no need to throw away all the bits before the synchronization is guaranteed (Theorems 3.15, 3.17 and 3.39).

The decoder may detect resynchronization faster, with use of Algorithm 3.1. If the algorithm answers that the decoder is synchronized at a bit number $p < K$, where K is the parameter of appropriate algorithm, the decoder further decodes correct data. However, if the method knownStartPosition is used, one has to take into account in Algorithm 3.1 that leaf strings are not codewords. Therefore, the first $h + \log N$ should be skipped.

Klein and Wiseman [38] presented a method for parallel Huffman decoding. It is based on dividing $\mathcal{E}$ into several blocks and decoding each block by a different processor. Due to the spontaneous synchronization of Huffman codes, the incorrectly decoded fragment at the beginning of a block (hereinafter called the block number B) is small. This fragment is decoded again by the processor that decoded the block $B-1$. In case the previous processor has not synchronized by the end of its block, the processor $B-2$ decodes the whole block $B-1$ and the beginning of block B, until it reaches the fragment, where the processor B has synchronized.

A disadvantage of this method is that in the worst case (no spontaneous synchronization) the work of all processors but the first one is wasted. The methods for guaranteed synchronization can be used in this case to guarantee that any decoder synchronizes quickly. This application will be discussed in Section 3.10.

The methods for guaranteed synchronization may be used to limit bit error propagation. If there are no insertion or deletion errors, the decoder always knows the

position of the currently decoded bits. The synchronization delay can thus be limited with the method KNOWNSTARTPOSITION. It has the advantage over the method UNKNOWNSTARTPOSITION of being simpler and faster. On the other hand, if insertions or deletions occur, only the method ANYSTARTPOSITION is appropriate.

Both methods for guaranteed synchronization do not guarantee strong synchronization. A decoder that starts from an arbitrary bit in $\mathcal{E}$ does not know which symbol of $\mathcal{M}$ it is decoding and does not know where to place the decoded symbols. The same problem appears when after a bit error wrong number of source symbols are decoded before the decoder resynchronizes. The following symbols are placed at wrong positions in $\mathcal{M}$ and, in some sense, the whole decoded message is corrupted.

To introduce strong synchronization one has to place some additional positional markers into the data. Such markers can be embedded for instance into $\mathcal{M}$. An advantage of introducing such markers in $\mathcal{M}$, instead of doing it on the level of $\mathcal{E}$, is that the markers are also compressed using a Huffman code, so their length is optimal.

Strong synchronization may also be achieved with the no-subword resynchronization marker, Section 3.7. The positional markers can be placed just after the no-subword markers. The decoder can correctly recognize all the insertions of the no-subword marker, so the positional markers following it are always correctly interpreted.

It should be noted that none of the methods presented here requires any modification of the Huffman code itself. The code remains optimal and there is no redundancy if no markers are inserted (but recall that in case of the no-subword resynchronization marker the marker itself has to be transmitted). The methods are thus particularly useful if markers are inserted rarely into the encoded message.

3.10 Applications to parallel decompression

As the number of CPUs in a personal computer is constantly increasing, it is getting more and more important to parallelize as much operation as possible. To decode compressed data in parallel the data has to be split into fragments, each of them to be processed separately. In such a division the following forces have to be taken into account:

- The amount of work in each fragment is not strictly proportional to the fragment's size. Some parts of data may be better compressed so more time is needed to decompress them, in particular to write the result.
- In a real computer system there is no guarantee that each processing thread gets equal amount of processor's time. Some threads may execute faster than others.

It follows that splitting compressed data into equal parts does not assure good load balancing and 100% utilization of all CPUs. A better approach is to divide the data into smaller parts and to create a pool of jobs with the number of jobs a few times greater than the number of processors. The more parts the data is split into, the better load balancing can be achieved.

Unfortunately, more parts require more time for coordination of work, which decreases the overall performance. To have both large fragments and good load balancing a dynamic approach to fragmenting can be used. In such an approach the fragments at the beginning of the data are large and are getting smaller and smaller as decoding proceeds towards the end. It should be noted that the number of parts and their size depend also on the number of CPUs used for decompression.

In most cases it is not possible to decode only a fragment of compressed data, so splitting compressed data into fragments has to be supported at the encoder. In case of Huffman codes, decoding a fragment of compressed data requires that the decoder is synchronized at the beginning of the fragment. With normal Huffman coding there is no bound on the synchronization delay, so some means of resynchronization inserted into the encoded message are desired ([38] presents a method for parallel decoding without guaranteed synchronization; see also Sections 1.2.7 and 3.9 for discussion). A few methods for the division of Huffman data into blocks with decoder synchronization are analyzed below.

The easiest method, hereinafter called WHOLECODEWORDS, is to assume a fixed block length, fill each block with as many codewords as possible, and pad the remaining space in each block with some extra bits. The easiest choice for the extra bits is a prefix of the next codeword, the one that did not fit entirely into the current block. A codeword that does not cross a block boundary is encoded normally. If a block boundary divides a codeword w into $w = w'w''$, the encoder encodes w' first, to close the previous block, and then encodes entire w in the new block.

This method uses at most $h - 1$ redundant bits per block, but in most cases the encoded codewords are shorter than h. It may be expected that the average redundancy is of the order of the average codeword length, averaged over the probability distribution of codewords[1]. With this method a processor can start decoding at the beginning of each block without any additional operations.

A little more involved method, hereinafter called LOGH, is to place $\lceil \log h \rceil$ bits at the beginning of each block. The bits tell the relative position of the beginning of the first codeword in the block. When a codeword w is to be encoded as $w'w''$, where w' fits at the end of some block and w'' is the remaining part, the encoder first outputs w', closing the block, then $|w''|$ in $\lceil \log h \rceil$ bits, and finally it outputs w''. Other codewords are encoded normally. The method uses a constant number, $\lceil \log h \rceil$, of redundant bits per block. The decoder at start of a block reads the first $\lceil \log h \rceil$ bits to get a value n, then skips the next n bits and continues normally.

Finally, the methods for guaranteed synchronization can be used for splitting Huffman-encoded data into blocks. The encoder simply uses the methods for guaranteed synchronization with synchronization delay's bound equal K. The decoder has to start K bits before the beginning of a block. Then, at start of the block, it is synchronized and is able to decode correct data. The best choice for the method of guaranteed synchronization is KNOWNSTARTPOSITION, because it is faster than the method UNKNOWNSTARTPOSITION and the position of each block is known to the

[1]This is just a guess, but there is no need to analyze the expected redundancy of WHOLECODEWORDS as the method KNOWNSTARTPOSITION, to be introduced shortly, performs much better.

decoder.

Note that in each of the three methods the actual parts processed by CPUs may be larger than one block. Blocks are smaller to increase the granularity of the division of work between CPUs. The granularity is important because the optimal fragment sizes are chosen dynamically by the decoder, and they are a function of the number of CPUs present and of the amount of data left.

Parallel decoding was tested on Jpeg compression. Jpeg is one of the most popular compression schemes that use Huffman codes. In Jpeg compression a codeword of a Huffman code is always followed by some number of bits that is determined by the encoded codeword. Such a code can be transformed into a larger Huffman code by attaching a full binary tree to each leaf of the initial Huffman tree. The height of the tree attached to a leaf is equal to the number of bits that follow the codeword of the leaf.

Jpeg files may contain a custom Huffman code, but in most cases the code being used is the one recommended by the standard. This code was also used for conducting the tests. The extended Huffman tree for the code of Jpeg is of height 26 and has about 32 thousand leafs. The code has 56 synchronizing codewords and the shortest synchronizing string is of length 18.

The tests were conducted on 10 grayscale images of size 3872x2592 — the typical size of pictures from modern digital cameras (10 MPix). The tests were run twice, for two values of the Jpeg quality factor, Q, which determines the compression ratio and the loss of quality. For $Q = 1$, which is the typical setting, the compressed files were of size about 0.5 MB. For $Q = 10$ the size was around 1.5 MB.

In case of Jpeg files, the processing time needed for decoding Huffman data is about an order of magnitude lower than the time needed for other stages of compression (e.g. DCT), so the tests focused on the redundancy introduced by the three methods of dividing encoded data into blocks. For large block sizes, B, the redundancy introduced by any of the three methods is negligible, so none of them is preferable.

The tests focused on very small block sizes, $B = 50, 100, 200, 500$ and 10000 bits, where the last one was much greater than the other ones in order to investigate how the methods behave if the block size is large. The choice of such a small block size is supported by the argumentation concerning the granularity of block division, presented before. It is also supported by the increasing number of processors in a personal computer. It is not unexpected that the number of CPUs in a PC raises to 100 in a decade.

The average redundancy introduced in the ten compressed files by each of the methods is shown in Table 3.13. The results for logH and wholeCodewords are about the same. WholeCodewords performs slightly worse for $Q = 10$ than for $Q = 1$ because a file compressed with such a quality factor contains more long codewords. The method logH, as expected, is insensitive to changes of Q.

The method knownStartPosition performs an order of magnitude better for $B = 50$ than the other two methods and two orders of magnitude better for $B = 200$. For $B = 10000$ and $Q = 1$ knownStartPosition needs exactly zero redundant bits for all ten images. For $B = 10000$ and $Q = 10$ only one image had a few bits inserted.

B	$Q = 1$			$Q = 10$		
	KSP	LogH	WC	KSP	LogH	WC
50	0.91%	11.07%	10.72%	0.84%	11.33%	13.29%
100	0.19%	5.24%	4.78%	0.11%	5.32%	6.30%
200	0.03%	2.56%	2.52%	0.01%	2.58%	3.03%
500	0.005%	1.01%	0.92%	0.001%	1.01%	1.18%
10000	0.000%	0.05%	0.04%	0.000%	0.05%	0.06%

Table 3.13: The redundancy introduced by dividing an encoded message into blocks of size B for the methods knownStartPosion (KSP), logH and wholeCodewords (WC). The results presented were computed for two values of the quality factor, Q.

No. CPUs	4-core	2-core
1	4.8	5.5
2	2.9	3.7
3	2.2	N/A
4	1.9	N/A

Table 3.14: Parallel decoding time is seconds for a Jpeg image encoded with knownStartPosition method, with 1, 2, 3 and 4 CPUs, tested on two different computers.

The advantages of the method knownStartPosition are clear. There are a few drawbacks concerning the overhead of this method terms of processing speed. It was presented in Section 3.8.3 that the overhead of knownStartPosition is from 16%, for small codes, to 60%, for large codes. The code in Jpeg is certainly large, but it has a particular structure, namely full subtrees attached to some internal nodes of the tree. This structure opens ways for optimizations.

Another disadvantage is that for each fragment the additional $B+h-1+\lfloor \log h \rfloor = B + 39$ bits have to be processed (see Theorem 3.15). If block sizes are small in comparison to fragments assigned to processors, the overhead is negligible, but for small fragments the overhead may be significant.

Parallel Jpeg decompression was also tested in terms of scalability with different number of processors used for decompression. The results are shown in table 3.14. The image was encoded with the knownStartPosition method with $K = 200$. For decoding, it was divided into 100 fragments of the same size. The decompression time is given in seconds and is an average from 10 runs. The tests were run on two different computers: the 4-core CPU was an Intel Core 2 Quad Q9300 processor (2.5GHz) and the 2-Core CPU was an Intel Core 2 Duo T7300 (2GHz).

The test should be seen as proof-of-concept of the method for parallel decoding rather than actual performance and scalability tests. The CPU utilization when running on 4 cores was about 75%, so the scalability may still be greatly improved.

The compressed images have also been tested for the maximum length of the synchronization delay. The results are presented in Table 3.15. In most cases the maximum synchronization delay is about 500-2000 bits, but for two files it exceeded 10,000. To protect the decoder from such synchronization delays, methods for guaranteed synchronization are desirable.

	max delay for $Q = 1$	max delay for $Q = 10$
File 1	1160	510
File 2	16687	1955
File 3	849	475
File 4	909	500
File 5	2269	1705
File 6	1159	557
File 7	1205	1592
File 8	1360	522
File 9	2595	603
File 10	7005	12390
Average	3520	2081

Table 3.15: Maximum synchronization delay, in bits, for the ten test Jpeg images for two values of the quality factor Q.

3.11 Conclusions

Important results of this chapter are the two novel methods for guaranteed synchronization of Huffman codes. The methods limit the delay of a decoder that starts at an arbitrary position in the encoded message. The first method, knownStartPosition (Section 3.4), requires that the decoder knows the position number of the bits being decoded. The other one, anyStartPosition (Section 3.6), does not.

Both methods do not require any modification of the Huffman code itself, so the code remains optimal. The methods provide any decoder with limited synchronization delay by inserting some markers into the message. The redundancy inserted depends on the synchronization properties of the code and on the message itself. It is much lower for codes with good synchronization properties. The redundancy is zero if spontaneous synchronization occurs fast enough because no markers are inserted in this case and also the code is optimal. Test have shown that the redundancy is of the order 0.01% for the synchronization delay bound equal approximately 500 bits. The methods can be applied, for instance, for limiting bit error propagation in encoded messages or for decoding a fragment of a Huffman encoded message.

Another new technique for improving error resilience of compressed data is the noSubword method for ensuring strong synchronization (Section 3.7). The method deals with the same problem as the extended synchronizing codeword [40], but it is in certain aspects superior. For instance, if no resynchronization markers are inserted into the message, the redundancy increases only logarithmically with the message size. In comparison, for ESC the growth is linear. Tests have shown that the resynchronization marker proposed here is, in most cases, shorter than ESC. The noSubword method is general enough to be applied to compression schemes other then Huffman codes.

Several results that help to understand better the spontaneous synchronization of Huffman codes have been presented. An important result of this kind is the method for tracking the state of several decoders at the same time. This topic has not been

considered before. Two algorithms were presented. The first one requires $O(h)$ operation per encoded symbol in the worst case. It is expected that on average, for typical codes, the time complexity is proportional to the length of the encoded message. The main advantage of the algorithm is its simplicity and the main drawback is the quadratic preprocessing time.

The other algorithm is more involved, but its time complexity is proportional to the length of the encoded message, in the worst case. The algorithms were used in the method for guaranteed synchronization with unknown start position. They can also be applied to computing the maximum synchronization delay for a decoder of a given message.

A synchronization graph for a Huffman code has been introduced. The graph can be used for testing whether a code has a synchronizing string, for the construction of a synchronizing string and for finding all synchronizing codewords of a code. It should be noted that the methods using the synchronization graph are not optimal, but are easy to understand. Better, but more involved methods are put forward in Chapter 4.

A very simple, yet practical result is the method for recognizing resynchronization by a decoder that started at some bit of the encoded message. This algorithm is also a novelty.

Applications of the new methods were discussed, in particular application to parallel decoding of Huffman-encoded data. Tests of such parallel decoding were performed on Jpeg images.

Chapter 4

Synchronizing strings for Huffman Codes

In this chapter topics related to synchronizing strings and codewords for Huffman codes are studied. In particular, several bounds on the length of the shortest synchronizing string for a synchronizing Huffman code are presented. Also, efficient algorithms for computing a short synchronizing string and for computing all the synchronizing codewords are introduced.

4.1 Merging string for a pair of states

Theorem 4.1. *Let C be a synchronizing Huffman code of size N, let T and $\mathcal{T}$ be, respectively, the Huffman tree and the Huffman automaton for C. For any node n of $\mathcal{T}$ there is a merging string s_n for the set $\{n, \varepsilon\}$ (ε is the root of T), with*

$$|s_n| \leq \sum_{p \in Q(T) \setminus \{\varepsilon\}} h_p, \tag{4.1}$$

where $Q(T)$ is the set of the internal nodes of T and h_p is the height of the subtree of T rooted at p.

Proof. C is synchronizing so $\mathcal{T}$ has a synchronizing string $s_{\mathcal{T}}$. Let us consider a merging string s_n for n and ε of minimal length. It exists, because $s_{\mathcal{T}}$ merges ε and n, but it need not be unique. The string s_n brings both nodes to the root, because otherwise we could remove the last letter of s_n and the result would still merge n and ε.

Let $\{n_i, m_i\}$ be the unordered pairs of nodes that appear when consecutive prefixes of s_n are applied to the initial configuration $\{n, \varepsilon\}$:

$$\{n_i, m_i\} = \delta^* \left(\{n, \varepsilon\}, s_n[..i) \right), \quad i = 0, \ldots, |s_n|. \tag{4.2}$$

In particular, $\{n_0, m_0\} = \{n, \varepsilon\}$ and $\{n_{|s_n|}, m_{|s_n|}\} = \{\varepsilon\}$ (a singleton is also considered as a pair).

Let us consider the subsequence $\{n_{i_k}, \varepsilon\}, k = 0, \ldots, l$, of this sequence, formed of pairs containing the root. Each node p, appears in this subsequence as the partner of ε at most once, because pairs do not repeat in $\{n_i, m_i\}$ (otherwise we could shorten the string s_n). The string $s_n[i_k, i_{k+1})$, that brings $\{n_{i_k}, \varepsilon\}$ to $\{n_{i_{k+1}}, \varepsilon\}$, is a string that

either brings the node n_{i_k} to a leaf without loops or that brings ε to a leaf without loops. In either case the length of $s_n[i_k, i_{k+1})$ is at most $h_{n_{i_k}}$ (note that in the second case the node $n_{i_{k+1}}$ is in the subtree of n_{i_k}, unless $n_{i_{k+1}} = \varepsilon$). It follows that

$$|s_n| = \sum_{k=0}^{l-1} |s_n[i_k, i_{k+1})| \leq \sum_{k=0}^{l-1} h_{n_{i_k}} \leq \sum_{p \in Q(T) \setminus \{\varepsilon\}} h_p. \tag{4.3}$$

The last inequality follows from the fact that n_{i_k} are different nodes of T. The value of h_ε is not counted in the sum because the set $\{\varepsilon\}$ appears only as the last element of the sequence $\{n_{i_k}, \varepsilon\}$. □

Let H_T be the value of the bound in Theorem 4.1. H_T is the sum of heights of all the nontrivial subtrees of T apart from the whole tree. We will compare H_T with Π_T — the sum of depths of all the internal nodes, with W_T — the sum of depths of all the leaves of T (that is the sum of codewords' lengths), and with S_T — the sum of sizes of all subtrees of T. The proof of Theorem 4.1 can be easily modified to prove that $|s_n|$ does not exceed Π_T, W_T and S_T. It turns out that the bound $|s_n| \leq H_T$ is the best of the four.

Lemma 4.2. *Let T be a complete binary tree, let $Q(T)$ be the set of the internal nodes of T, let $L(T)$ be the set of leaves of T, let h_n and N_n be, respectively, the height and the number of leaves in the subtree rooted at the node n of T, let $|\pi(n)|$ be the distance from the root to n and let N_T be the number of leaves in T. Let us define:*

$$H_T = \sum_{n \in Q(T) \setminus \{\varepsilon\}} h_n, \qquad \Pi_T = \sum_{n \in Q(T)} |\pi(n)|, \tag{4.4}$$

$$W_T = \sum_{n \in L(T)} |\pi(n)|, \qquad S_T = \sum_{n \in Q(T) \cup L(T)} N_n. \tag{4.5}$$

Then the following holds:

$$\Pi_T = W_T - 2N_T + 2, \tag{4.6}$$

$$W_T = S_T - N_T, \tag{4.7}$$

$$H_T \leq \Pi_T \leq W_T \leq S_T. \tag{4.8}$$

Proof. We prove the equations and the inequalities by induction. For a tree that consist of just the root node the values H_T, Π_T and W_T are equal 0, S_T and N_T is equal 1 and (4.6), (4.7), (4.8) are correct. If the tree T consists of two subtrees T_1 and T_2 joined in a common root, the following recurrences hold:

$$H_T = H_{T_1} + H_{T_2} + h_{T_1} + h_{T_2}, \tag{4.9}$$

$$\Pi_T = \Pi_{T_1} + \Pi_{T_2} + (N_{T_1} - 1) + (N_{T_2} - 1), \tag{4.10}$$

$$W_T = W_{T_1} + W_{T_2} + N_{T_1} + N_{T_2}, \tag{4.11}$$

$$N_T = N_{T_1} + N_{T_2}, \tag{4.12}$$

$$S_T = S_{T_1} + S_{T_2} + N_{T_1} + N_{T_2}, \tag{4.13}$$

where h_{T_i} is the height of the tree T_i.

Equation (4.6) can be proved by induction, by substituting the induction hypothesis (4.6) for Π_{T_1} and Π_{T_2} to (4.10) and using (4.11) and (4.12):

$$\begin{aligned}\Pi_T &= \Pi_{T_1} + \Pi_{T_2} + N_{T_1} + N_{T_2} - 2\\ &= (W_{T_1} - 2N_{T_1} + 2) + (W_{T_2} - 2N_{T_2} + 2) + N_{T_1} + N_{T_2} - 2\\ &= (W_{T_1} + W_{T_2} + N_{T_1} + N_{T_2}) - 2(N_{T_1} + N_{T_2}) + 2\\ &= W_T - 2N_T + 2.\end{aligned}$$

The recurrences for W_T and $S_T - N_T$ are identical:

$$S_T - N_T = (S_{T_1} - N_{T_1}) + (S_{T_2} - N_{T_2}) + N_{T_1} + N_{T_2} \tag{4.14}$$

and the value of $(S_T - N_T)$ for the tree with one node is 0, so $S_T - N_T = W_T$ holds for all trees. The inequality $h_T \le N_T - 1$ together with the recurrences (4.9) and (4.10) prove the first inequality of (4.8). Other inequalities follow from (4.6) and (4.7). □

Corollary 4.3. *Let w_i be all the codewords of a Huffman code. Then*

$$|s_n| \le \sum_i |w_i| \qquad \textit{and} \qquad |s_n| \le (N-2)(h-1). \tag{4.15}$$

The result of Theorem 4.1 can be improved if we notice that the sequence $\{n_{i_k}, \varepsilon\}$, defined in the proof of Theorem 4.1, cannot contain two nodes n_{i_k} and $n_{i_{k'}}$ that are roots of identical subtrees of T. Indeed, otherwise we could shorten the string s_n in the same way as before. This gives the following result:

Corollary 4.4. *The bound of Theorem 4.1 can be improved to:*

$$|s_n| \le \sum_{t \in \mathscr{T}(T) \setminus \{T\}} h_t, \tag{4.16}$$

where $\mathscr{T}(T)$ is the set all distinct subtrees of T and h_t is the height of the tree t.

Example 4.5: Let us consider the tree for the code C_1 from Figure 2.3. It has two distinct subtrees other than the whole tree: one of height 1 and one of height 2. It follows that for each node n there is a merging string with ε of length at most 3. For instance, for the node 0 the merging string is 110.

The tree for the code C_2 from Figure 2.5, has exactly the same subtrees, so the bound on the length of its merging strings is the same. In this case the node 01 requires a merging string of length at least 3, for instance 011.

Note that the bound is not sharp and for some trees all nodes have shorter merging strings. For instance, each node of the tree from Figure 4.1 can be merged with the root using a string of length 1 (the string '1'). The bound from Corollary 4.4 for this tree is 3.

The idea of identifying common subtrees can be formalized by introducing the *minimized Huffman automaton*. Although this does not give here a better estimate of the length of the shortest merging string for the set $\{n, \varepsilon\}$, it is interesting in itself. This construction will also be used later in this chapter.

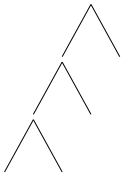
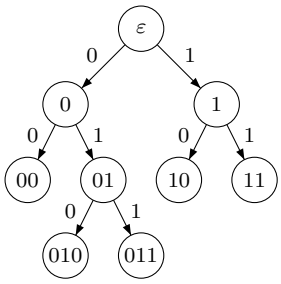
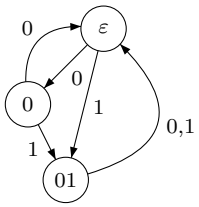


Figure 4.1: A tree for which 1 is a merging string for any set of nodes.



(a) Huffman tree automaton for the code C_2.



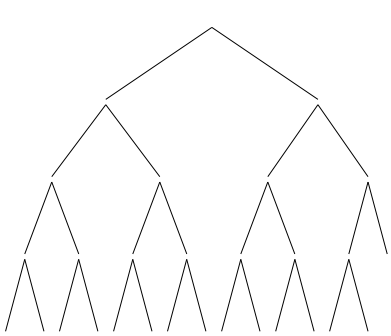
(b) Minimized Huffman automaton for the code C_2. The states '1' and '01' were merged into the state 01.

Figure 4.2: A Huffman automaton and its minimized version.

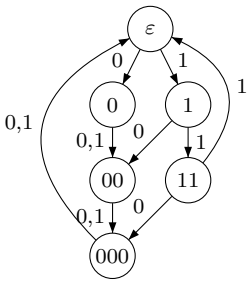
Definition 4.6. A *minimized Huffman automaton* for a Huffman code C is an automaton made of the Huffman automaton for C by merging the states that are roots of identical subtrees of the Huffman tree T for C.

It is easy to see that minimized Huffman automata have exactly two edges, labeled with 0 and 1, going out of each node, thus their transition function is indeed a function. Examples of minimized Huffman automata are presented in Figures 4.2(b) and 4.3(b).

We will say that a set V of states of a Huffman automaton $\mathcal{T}$ corresponds to the set V_m of states of the minimized Huffman automaton $\mathcal{T}_m$ if V_m is the smallest set satisfying: if $q \in V$ and q is merged to a state q' of $\mathcal{T}_m$ then $q' \in V_m$.



(a) Huffman tree



(b) Minimized Huffman automaton

Figure 4.3: A Huffman tree and its minimized Huffman automaton.

Theorem 4.7. *Let C be a synchronizing Huffman code, let $\mathcal{T}$ be the Huffman automaton for C and let $\mathcal{T}_m$ be the minimized Huffman automaton for C. Let V be a set of states of $\mathcal{T}$ and let V_m be the corresponding set of states of $\mathcal{T}_m$. If s is a merging string for V then s is a merging string for V_m. If s' is a merging string for V_m that brings all nodes of V_m to the root then s' is a merging string for V.*

Proof. Let us consider coins on the states of the automata $\mathcal{T}$ and $\mathcal{T}_m$. Moving coins according to a string w and then merging the states of $\mathcal{T}$ to get its minimized version (with removing duplicate coins on the same state) is equivalent to merging the states first and then moving coins. This observation proves the theorem. Indeed, applying s to V leaves only one coin. Then merging the states does not multiply the coins. Applying s' to V_m leaves only the coin in the root. Thus, applying s' to V may leave a coin only in the root of $\mathcal{A}$, because no other state was merged with the root of $\mathcal{T}$ to be the root of $\mathcal{T}_m$. □

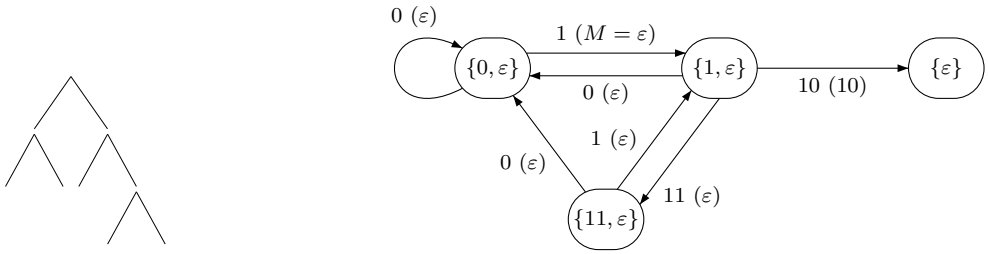
Note that the minimized Huffman automaton is implicitly used in Corollary 4.4, where we consider all non-identical subtrees of a Huffman tree. The roots of such subtrees are states of the minimized Huffman automaton.

Theorem 4.1 leads to an algorithm for finding the shortest merging string for a set $\{n_0, \varepsilon\}$, where n_0 is any state of $\mathcal{T}$. First a graph $G = (V, E)$ is created. The vertices of G are unordered pairs $\{n, \varepsilon\}$, where n is a state of $\mathcal{T}$. The edges of G are weighted; $\{n_1, \varepsilon\} \to \{n_2, \varepsilon\}$ is an edge if there is a string w that brings $\{n_1, \varepsilon\}$ to $\{n_2, \varepsilon\}$ without passing through any other pair $\{n, \varepsilon\}$. The weight of the edge is the length of the shortest such string w (note that the string w need not be unique).

Such a string w is also the label of the edge $\{n_1, \varepsilon\} \to \{n_2, \varepsilon\}$, although it will not be stored explicitly. Instead, for retrieving the label w, we will store a mark M. The mark will depend on the target pair of the edge. If the target is a pair $\{n_2, \varepsilon\}$ with $n_2 \neq \varepsilon$, the mark is equal to either n_1 if $n_2 = \delta(n_1, w)$, or to ε if $n_2 = \delta(\varepsilon, w)$. In either case $n_2 = \delta(M, w)$, the node n_2 is in the subtree of the node M and w is formed of labels on the path from M to n_2. If the target of an edge is a singleton $\{\varepsilon\}$, that is $n_2 = \varepsilon$, the mark M is the leaf $\delta(\varepsilon, w)$. In this case the word w is formed of labels on the path from ε to M.

The construction of the graph requires DFS-traversing the Huffman tree with a pair of nodes $\{n_1, n_2\}$, starting at $\{n, \varepsilon\}$ and applying transitions of the Huffman automaton to both nodes of the pair. The traversal goes forward until a set $\{n', m\}$ is reached, with m being a leaf. Then, the edge $\{n, \varepsilon\} \to \{n', \varepsilon\}$ is added to the graph with the number of steps from $\{n, \varepsilon\}$ to $\{n', m\}$ as its weight. If such an edge has been added before, only the weight is updated to be the minimum of the previous weight and the new one. Finally, the mark M of the edge is set appropriately.

The cost of processing each pair $\{n, \varepsilon\}$ during the construction of the graph G is proportional to the size of the subtree rooted at n, because the DFS traversal is limited to the subtree of n. It follows that the construction of G uses the time proportional to the sum of sizes of the subtrees of T, S_T. By Lemma 4.2, this is $O(\sum |w_i|)$, where w_i are the codewords given by the tree T. The number of vertices in the graph is $|V| = N - 1$. The number of edges is bounded by the sum of sizes of all the subtrees of the tree, that is $|E| = O(\sum |w_i|)$.



(a) Huffman tree for the code C_1

(b) The graph from the algorithm for finding the shortest merging strings for the code C_1; the mark M is given in parenthesis

Figure 4.4: Illustration for the algorithm for finding the shortest merging strings.

The shortest merging string for a set $\{n, \varepsilon\}$ is given by the lightest path from $\{n, \varepsilon\}$ to $\{\varepsilon\}$ in the graph G. The tree of the lightest paths from any node to $\{\varepsilon\}$ can be constructed using Dijkstra's algorithm in $O(|E| + |V| \log |V|)$. Since $|V| = O(N)$, $|E| = O(\sum |w_i|)$ and $\sum |w_i| \geq N \log N$, the lightest paths' tree can be computed in $O(\sum |w_i|)$.

To print out the shortest merging string for a set $\{n, \varepsilon\}$ it is necessary to reconstruct the labels of each edge. For an edge $\{n, \varepsilon\} \to \{n', \varepsilon\}$, where $n' \neq \varepsilon$, we may traverse the tree up from n' until we reach the node M (the mark of the edge). The string w is formed of labels on the path that goes from M down to n'. The word can be computed in the time proportional to its length. For an edge $\{n, \varepsilon\} \to \{\varepsilon\}$ we do the same, but we traverse the tree from M up to ε.

The description of the algorithm can be summarized as follows.

Theorem 4.8. *Let $\mathcal{T}$ be a Huffman automaton. The algorithm for computing the shortest merging string for a set $\{n, \varepsilon\}$, where n is any state of $\mathcal{T}$, requires $O(\sum_i |w_i|)$ preprocessing time. Then, the shortest merging string for each $\{n, \varepsilon\}$ pair can be found in time proportional to the length of the merging string.*

Example 4.9: Let us see how the algorithm works for the code C_1 from Figure 2.3. Figure 4.4 presents the graph for this code. There are four internal nodes in the tree, so there are four nodes in the graph. Each edge in the graph is marked with the word w that brings a state to the other one, and also with the mark M, in parenthesis. Shortest merging strings can easily be read from the graph. For the set $\{0, \varepsilon\}$ it is 110, for $\{1, \varepsilon\}$ — 10, and for $\{11, \varepsilon\}$ — 110.

4.2 Length of a synchronizing string

In this section we give an upper bound on the length of the shortest synchronizing string for any synchronizing Huffman code. We begin with a lemma that helps to prove the main theorem of this section (Theorem 4.12).

Lemma 4.10. *Let T be a complete binary tree with N leaves. There exists a string w of length at most $\lceil \log N \rceil$ such that for each node n of T some prefix of w labels a path from n to a leaf.*

Proof. Let us assume the contrary: for any string w of length $|w| = \lceil \log N \rceil$, there is a node n_w such that no prefix of w brings it to a leaf. Let $m_w = \delta(n_w, w)$. Then m_w is an internal node of the tree and w is a suffix of $\pi(m_w)$.

For two strings, $w_1 \neq w_2$, of length $\lceil \log N \rceil$ the nodes m_{w_1} and m_{w_2} are different. Indeed, the suffixes of $\pi(m_{w_1})$ and $\pi(m_{w_2})$ of length $\lceil \log N \rceil$ are different as they are equal w_1 and w_2, respectively. But there are $N-1$ internal nodes of T and at least $2^{\lceil \log N \rceil} \geq N$ strings w. The contradiction proves that the initial assumption is wrong. □

Example 4.11: Lemma 4.10 states that there is a string of length at most $\lceil \log 5 \rceil = 3$ that moves each node of the tree for the code C_1 (Figure 4.4(a)) through a leaf. Indeed, the string 00 is such a string.

Theorem 4.12. *For any synchronizing Huffman code of size N the length of the shortest synchronizing string s is at most*

$$|s| \leq \lceil \log N \rceil + (\lceil \log N \rceil - 1)X = O(Nh \log N), \tag{4.17}$$

where h is the length of the longest codeword, and

$$X = \sum_{t \in \mathscr{T}(T) \setminus \{T\}} h_t, \tag{4.18}$$

and $\mathscr{T}(\mathcal{T})$ is the set all different subtrees of T, and h_t is the height of the tree t.

Proof. Let us consider the Huffman automaton $\mathcal{T}$ for the code. $\mathcal{T}$ is synchronizing because the Huffman code is synchronizing. We will look at the power automaton for $\mathcal{T}$ by considering coins on the states of $\mathcal{T}$. Let Q be the set of all states of $\mathcal{T}$. Initially, there is a coin on every state of Q. A synchronizing string is a string that brings all the coins to the same state. Such a string, s, will be constructed by concatenating some words.

Let a coin c be on an internal node n of the tree and let a string w be applied to c. If c passes through a leaf, its path can be split into three parts: $w = w_1 w_2 w_3$;

- w_1: going down the tree until c reaches a leaf,
- w_2: going several times from the root down to a leaf and reappearing in the root,
- w_3: going down the tree from the root without passing through a leaf,

where w_2 and w_3 may be empty. Formally, w_1 is a nonempty suffix of some codeword, w_2 is a string of codewords, and w_3 is a proper prefix of some codeword. The word w_3 is also a proper suffix of w. The final position of c is fully determined by the part w_3, which, on the other hand, is fully determined by w and the length of w_3.

The first part of the synchronizing string is the word w given by Lemma 4.10. We have $|w| \leq \lceil \log N \rceil$ and after applying w any coin will pass through a leaf. The final position of any coin corresponds to some proper suffix of w. There are at most $\lceil \log N \rceil$ proper suffixes, because $|w| \leq \lceil \log N \rceil$. The set $A = \delta(Q, w)$ contains nodes that have a coin after applying w to the initial configuration, Q. Then $|A| \leq \lceil \log N \rceil$. We may further assume that $\varepsilon \in A$, because otherwise we can shorten the string w and get a set of the same size.

The automaton $\mathcal{T}$ is synchronizing, so, by Corollary 4.4, for each node n there is a string w_n of length at most X that merges ε and n. The final part of the synchronizing string for the code will be constructed from the strings w_n. Let

$$U_0 = A \setminus \{\varepsilon\}. \tag{4.19}$$

Let n_1 be any node from U_0. Let

$$U_1 = \delta(U_0, w_{n_1}) \setminus \{\varepsilon\}. \tag{4.20}$$

Then $|U_1| \leq |U_0| - 1$, because $\delta(n_1, w_{n_1}) = \varepsilon$ and ε is removed from U_1. Let us choose any n_2 from U_1 and let

$$U_2 = \delta(U_1, w_{n_2}) \setminus \{\varepsilon\}. \tag{4.21}$$

We can continue this process by setting

$$U_j = \delta(U_{j-1}, w_{n_j}) \setminus \{\varepsilon\}, \quad \text{for } j > 0. \tag{4.22}$$

In each step the size of U_j decreases by at least one. The process has to end at some k with $U_k = \emptyset$. Then $k \leq |U_0| \leq \lceil \log N \rceil - 1$. It easy to verify that $s = w w_{n_1} w_{n_2} \dots w_{n_k}$ brings any coin to the root, which means that it synchronizes the automaton $\mathcal{T}$. The bound on the length of s can be computed by estimating the length of each of its parts and the number of words w_{n_i}:

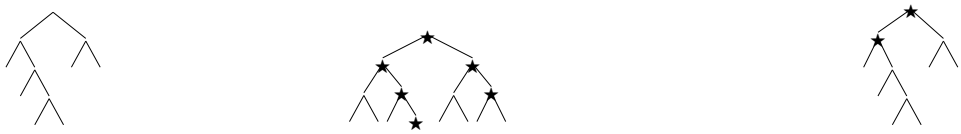
$$|s| \leq \lceil \log N \rceil + (\lceil \log N \rceil - 1)X. \tag{4.23}$$

Finally, the value of X is $O(Nh)$ because the sum has at most N terms and each term is less than h. □

The proof of Theorem 4.12 is constructive and gives an algorithm for the construction of a synchronizing string for a Huffman code. The algorithm works as follows.

First, a string w from Lemma 4.10 is found. It is done by checking all the $O(N)$ strings of length less or equal $\lceil \log N \rceil$ in the following way. For each node n of T, its subtree is DFS-traversed. Each time a node m that is not farther than $\lceil \log N \rceil$ steps below n is reached, the string w on the path from n to m is marked as BAD. This means that no prefix of w brings n to a leaf. The set of all strings of length less or equal $\lceil \log N \rceil$ can be stored in a full binary tree of height $\lceil \log N \rceil$

After the traversal, the strings that have not been marked as BAD bring any node through a leaf. By Lemma 4.10, there is at least one such a string of length $\lceil \log N \rceil$ or less. In fact, with this approach the shortest such string is found. The cost of this phase is proportional to the sum of sizes of all subtrees of T, S_T, which is $O(\sum_i |w_i|)$.



(a) A tree

(b) Strings of length ≤ 3 presented on a full binary tree; the strings that do not bring all nodes of the tree from Figure 4.5(a) through a leaf are marked with a star

(c) The result of applying the string 00 to the set of all states of the tree

Figure 4.5: Illustration for the algorithm of finding a synchronizing string for a code.

After finding the string w we can apply it to the set of all internal nodes of T. This takes $O(N \log N)$ time. Then, at most $\log N$ merging strings for $\{n, \varepsilon\}$, with some node n, suffice to build a synchronizing string. Computing the merging strings requires $O(\sum_i |w_i|)$ preprocessing time and then any string can be read in the time proportional to its length (Theorem 4.8). The length of each merging string is bounded by X and there are at most $\log N$ vertices that have to be moved using each such string. Thus the total cost of the algorithm is $O(X \log^2 N + \sum_i |w_i|)$.

Theorem 4.13. *Let C be a Huffman code of size N. The time complexity for the algorithm that computes a synchronizing string for C is*

$$O(X \log^2 N + \sum_i |w_i|), \tag{4.24}$$

where X is defined as in Theorem 4.12 and w_i are codewords of C.

Example 4.14: Let us consider the tree from Figure 4.5(a). There should be a string of length at most 3 that brings all nodes through a leaf. Such a string can be found with the algorithm just described. Figure 4.5(b) shows the tree of strings of length at most 3. The strings that were marked as BAD are starred. The shortest nodes that are not starred are 00 and 10. Both strings bring all nodes through a leaf. Let us chose 00.

Figure 4.5(c) shows the coins that remain after applying the string 00 to the tree with coins on all states. These two nodes with coins can be merged with a string of length at most 6, according to Corollary 4.4. In fact, the string 10 is the shortest merging string for these two nodes. The synchronizing string for the tree found by the algorithm is 0010. This is not the shortest synchronizing string, for instance 010 is a shorter synchronizing string for this tree.

From the proof of Theorem 4.12 it follows that if for each pair $\{n, \varepsilon\}$ there is a merging string then the code has a synchronizing string. This also gives an algorithm to test whether a synchronizing string for a code exists, because the existence of the merging strings can be checked with the algorithm of Section 4.1.

Theorem 4.15. *The complexity for the algorithm for testing if a code has a synchronizing string is $O(\sum_i |w_i|)$.*

4.3 Worst-case trees

Numerical search was performed to find:

- the worst-case trees in terms of the length of the shortest synchronizing string,
- the worst-case trees in terms of the length of the shortest merging strings for a pair $\{n, \varepsilon\}$, where n is an internal node of the tree.

All trees of sizes, N, from 3 to 20 were analyzed first. Then the procedure was repeated for all trees of heights, h, from 2 to 5.

4.3.1 Long synchronizing string

The worst-case trees for a fixed number of nodes, N, for $N = 3 \dots 12$, are shown in Figure 4.6.

In most of the tested cases the worst-case trees for fixed N, were unique up to the reflection across the y axis (relabeling 0-edges to 1-edges and 1-edges to 0-edges). The exceptions were the trees with 7 nodes — three nonequivalent trees, 10 nodes — 5 trees, and 12 nodes — 2 trees. For trees with 9, 11 and from 13 to 20 nodes the unique worst-case tree corresponds to one of the codes C_k, defined below. The codes C_k also form one of the worst-case trees with 7, 10 and 12 nodes. They can be described by the following set of codewords:

$$C_k = \{00, 010, 011, 110, 111\} \cup \{10^i 1 | i = 1, 2, \dots, k-1\} \cup \{10^k\}, \quad k \geq 1. \quad (4.25)$$

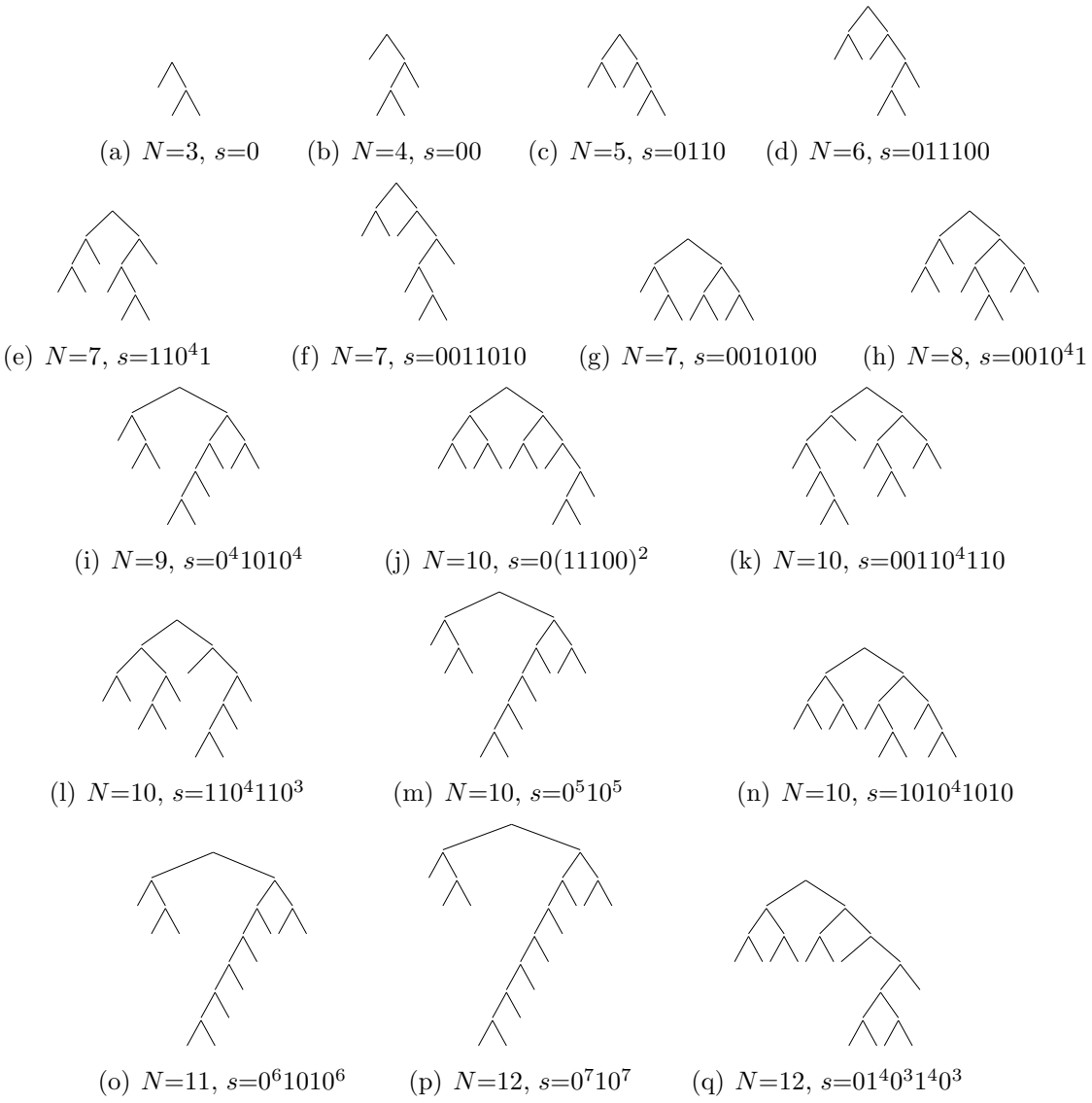
The size of the code C_k is $k+5$. The structure of these trees is shown in Figure 4.7(a) and examples can be found in Figures 4.6(g), 4.6(i), 4.6(m), 4.6(o), 4.6(p).

Theorem 4.16. *The shortest synchronizing string for the tree C_k, $k \geq 1$, is $s_0 = 0^k 10^k$ for odd k (even number of codewords) and $s_1 = 0^k 1010^k$ or $s_2 = 0^k 1110^k$ for even k (odd number of codewords). The length of the shortest synchronizing string is $2N - 9$ for even code size, N, and $2N - 7$ for odd code size.*

Proof. We will first find a merging string for the set of nodes A $= \{\epsilon, 1, 11\}$. We will look at the power automaton by placing coins on states of the automaton $\mathcal{T}$ for the code. Assume for a moment that $k > 3$. The transitions of coins under the letters 0 and 1 for selected three-coin configurations are presented in Figure 4.8. Configuration A is the initial configuration. All strings that lead to a configuration with less than two coins have to pass through the configurations B, C and E. The first configuration with two coins that may appear after starting from A is L $= \{\epsilon, 0\}$. The shortest string leading from A to L is 0^k and it is unique.

Figure 4.9 shows a fragment of the power automaton with two-coin configurations. The strings that reduce the number of coins to one have to pass through either N, for odd k, or through P, for even k.

The shortest string leading from L to N is 1000 and it is unique. Then, for odd k, the word 0^{k-3} merges the two coins of the configuration N. The shortest merging string for the configuration A is then $s_0 = 0^k 10^k$.



(a) N=3, s=0 (b) N=4, s=00 (c) N=5, s=0110 (d) N=6, s=011100

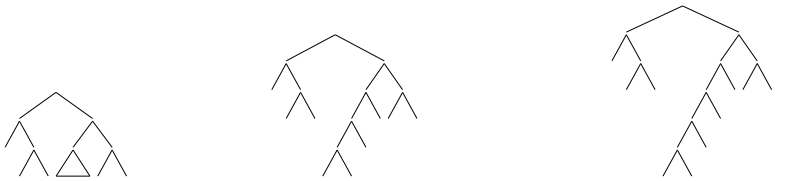
(e) N=7, s=110^41 (f) N=7, s=0011010 (g) N=7, s=0010100 (h) N=8, s=0010^41

(i) N=9, s=0^41010^4 (j) N=10, s=$0(11100)^2$ (k) N=10, s=00110^4110

(l) N=10, s=110^4110^3 (m) N=10, s=0^510^5 (n) N=10, s=1010^41010

(o) N=11, s=0^61010^6 (p) N=12, s=0^710^7 (q) N=12, s=$01^40^31^40^3$

Figure 4.6: Trees with longest synchronizing string for a given number of nodes, N. The synchronizing string is denoted by s.



(a) A tree C_k (b) N=9, s=$0^4 1010^4$ (c) N=10, s=$0^5 10^5$

Figure 4.7: The class C_k, of the worst-case trees in terms of the length of the shortest synchronizing string for a given number of nodes, N. The synchronizing string is denoted by s. The triangle denotes a code $\{1, 01, 001, \ldots, 0^i 1, 0^i 0\}$, $i = 0, 1, \ldots$.

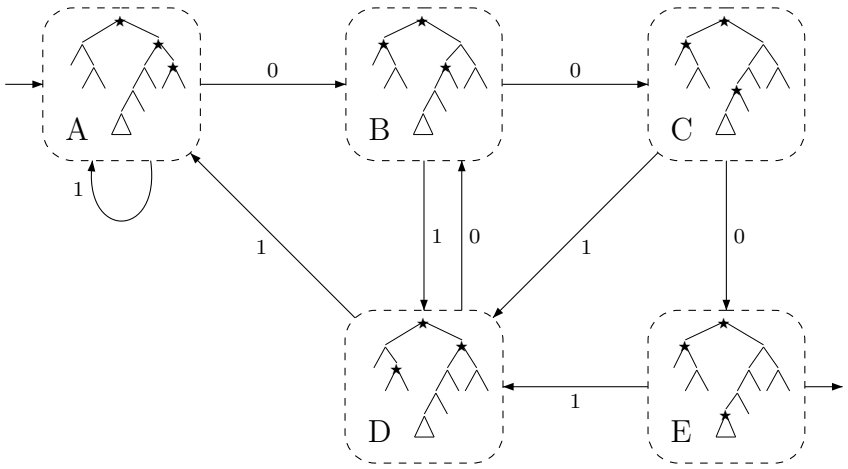


Figure 4.8: A fragment of the power automaton for the trees C_k. The coins on nodes of the automaton are marked with a star. Only selected three-state configurations are shown.

There are two shortest strings from L to P: 11100 and 10100. Then, for even k, the two coins of P can be merged with 0^{k-2}. The shortest merging strings for the configuration A are, in this case, $s_1 = 0^k 1010^k$ and $s_2 = 0^k 1110^k$.

The proof is correct so far for $k > 3$, but it is easy to verify that s_0 is the shortest merging string for the configuration A for $k = 1$ and $k = 3$, and that s_1 and s_2 are the shortest merging strings for A if $k = 2$.

To finish the proof it is necessary to show that s_0 is a synchronizing string for C_k if k is even, and that s_1 and s_2 are synchronizing strings for C_k if k is odd. It is easy to see that applying 0^k, which is a prefix of each of the strings s_0, s_1 and s_2, to the configuration with coins on all states always results in the configuration L, no matter if k is even of odd. This is the same as applying 0^k to A, so these strings synchronize C_k.

The length of the synchronizing string follows easily from its form and from $k = N - 5$. □

The worst-case trees for fixed height, h, with $h = 2, 3, 4$ and 5, are shown in Figures 4.10(a), 4.10(b), 4.10(c) and 4.10(d). These are full binary trees with two edges in the lower-right corner removed. They can be described by the following set

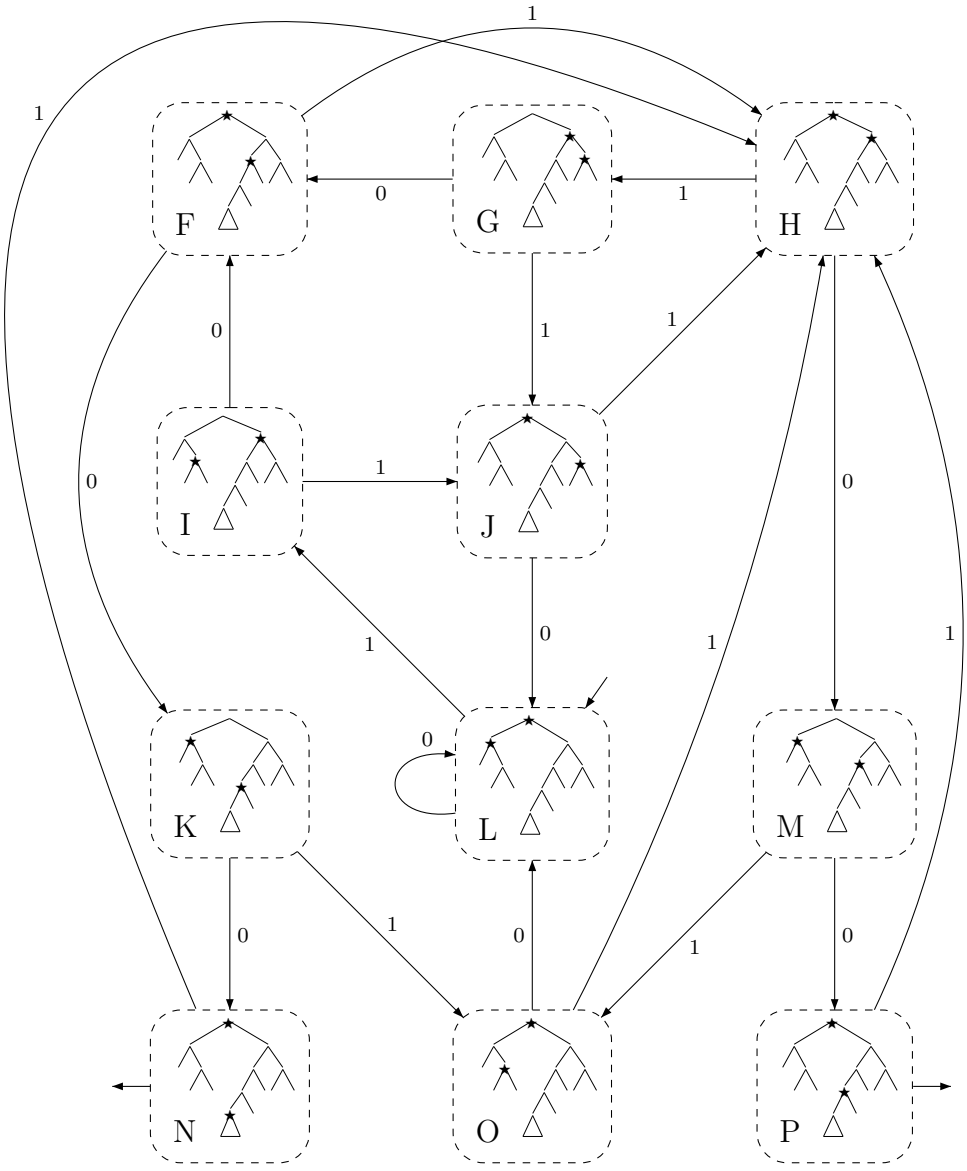
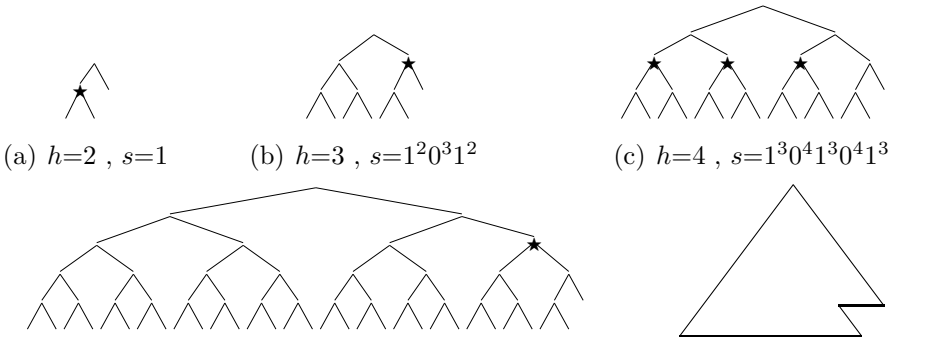


Figure 4.9: A fragment of the power automaton for the trees C_k. Coins are marked with a star. Only selected two-coin configurations are shown.



(a) h=2 , s=1 (b) h=3 , s=$1^2 0^3 1^2$ (c) h=4 , s=$1^3 0^4 1^3 0^4 1^3$

(d) h=5 , s=$1^4 0^5 1^4 0^5 1^4 0^5 1^4$ (e) The scheme for D_h

Figure 4.10: Trees with the worst-case length of a synchronizing and merging string among trees of fixed height, h, for $h = 2, 3, 4, 5$, and a scheme of these trees. The nodes n with the longest merging string for $\{n, \varepsilon\}$ are marked with a star.

of codewords:

$$D_h = \left(\{0,1\}^h \setminus \{1^{h-1}1, 1^{h-1}0\}\right) \cup \{1^{h-1}\}, \quad h \geq 2. \tag{4.26}$$

The general scheme for the trees D_h is depicted in Figure 4.10(e). The number of codewords in the code D_h is $2^h - 1$. The trees D_h are unique worst-case trees up to the reflection across the y axis.

Theorem 4.17. *The shortest synchronizing string for the tree D_h, $h \geq 2$, is*

$$s = (1^{h-1}0^h)^{h-2}1^{h-1}, \tag{4.27}$$

with $|s| = 2h^2 - 4h + 1$ (however, the shortest synchronizing string is not unique).

The proof of Theorem 4.17 is long and will be split into several steps.

First, we will find the shortest merging string s_{L_h} for the set of nodes on the leftmost path of the tree, i.e. for the set $L_h = \{\epsilon, 0, 00, \ldots, 0^{h-1}\}$. No synchronizing string for the tree D_h can be shorter than s_{L_h}. We will later show that s_{L_h} is also a synchronizing string for D_h.

Let $\mathcal{F}_h$ be the minimized Huffman automaton for D_h, constructed from the automaton for D_h by merging states with the same subtrees. Let L'_h be the set of states of $\mathcal{F}_h$ that corresponds to L_h. From Lemma 4.17 we know that if s is a merging string for the set L'_h in the automaton $\mathcal{F}_h$ and if s moves the coins to the root, it is also a merging string for the set L_h in the automaton D_h.

The automaton $\mathcal{F}_5$ for the tree D_5 is depicted in Figure 4.11 (see also Figure 4.10(d)).

It is easy to see that the general automaton $\mathcal{F}_h$, $h \geq 3$, has nodes $R, A_1, \ldots, A_{h-1}$ and $B_1, \ldots, B_{h-2}$ (see Figure 4.11) and its transitions are similar to the ones of $\mathcal{F}_5$.

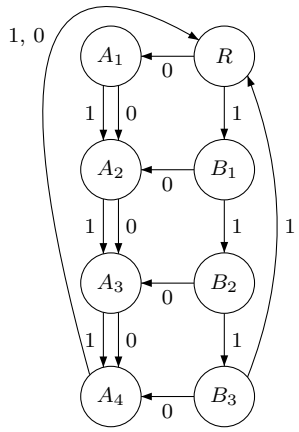


Figure 4.11: The automaton $\mathcal{F}_5$ equivalent to the tree D_5.

The nodes of $\mathcal{F}_h$ correspond to the following nodes of the Huffman tree.

$$R \to \{\epsilon\}$$

$$A_1 \to \{0\}$$
$$A_2 \to \{00, 01, 10\}$$
$$\dots$$
$$A_i \to \{0+1\}^i \setminus \{1^i\}, \quad (i = 1 \dots, h-1)$$
$$\dots$$

$$B_1 \to \{1\}$$
$$B_2 \to \{11\}$$
$$\dots$$
$$B_j \to \{1^j\}, \quad (j = 1, \dots, h-2),$$
$$\dots$$

The set L'_h is equal $\{R, A_1, A_2, \dots, A_{h-1}\}$.

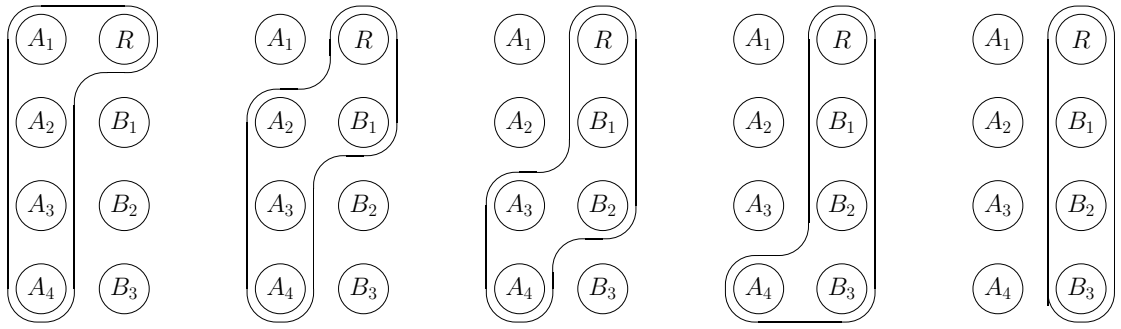
Let us fix h. Let the sets (configurations) β_i of states of $\mathcal{F}_h$ be defined by:

$$\beta_i = \delta^*_{\mathcal{F}_h}(L'_h, 1^i), \quad i = 0, 1, \dots, h-1, \tag{4.28}$$

where $\delta_{\mathcal{F}_h}$ is the transition function for the automaton $\mathcal{F}_h$. The sets $\beta_0, \dots, \beta_4$ for the automaton $\mathcal{F}_5$ are shown in Figure 4.12. We see that

$$\delta_{\mathcal{F}_h}(\beta_i, 1) = \beta_{i+1} \quad \text{for } i = 0, \dots, h-2, \tag{4.29}$$
$$\delta_{\mathcal{F}_h}(\beta_{h-1}, 1) = \beta_{h-1}. \tag{4.30}$$

Also $\delta_{\mathcal{F}_h}(\beta_i, 0) \subseteq \beta_0$ for any $i < h$ (for $i < h-1$ it is even $\delta_{\mathcal{F}_h}(\beta_i, 0) = \beta_0$). Let us consider a configuration S. If $S \subseteq \beta_i$ for some i then both $\delta_{\mathcal{F}_h}(S, 0)$ and $\delta_{\mathcal{F}_h}(S, 1)$ are



(a) β_0 (b) β_1 (c) β_2 (d) β_3 (e) β_4

Figure 4.12: The configurations β_i for the automaton $\mathcal{F}_5$.

subsets of some β_j and β_k. By induction, for all strings s, $\delta_{\mathcal{F}_h}(L'_h, s) \subseteq \beta_i$ for some i (but i need not be unique).

Now, let us consider a merging string s_h for the set L'_h of $\mathcal{F}_h$. If s_h has a substring 01^i0 with $i < h-1$, this substring can be substituted by 00^i0 and the resulting string is still merging for L'_h. Indeed, after a 0 the automaton is in a configuration C that is a subset of β_0. Then the strings 1^i0 and 0^i0 both bring it to exactly the same configuration $C' \subseteq \beta_0$. If we denote $A_0 = R$, the coin from A_k goes in both cases to $A_{(k+i+1) \bmod h}$ (see also Figures 4.11 and 4.12).

As a result, if s is a merging string for the set L'_h of $\mathcal{F}_h$ then there is a merging string s' for L'_h of the same length with no substrings of the form 01^i0, $i < h-1$. We may also assume that the string s' does not start with 1^i0, $i < h-1$ either.

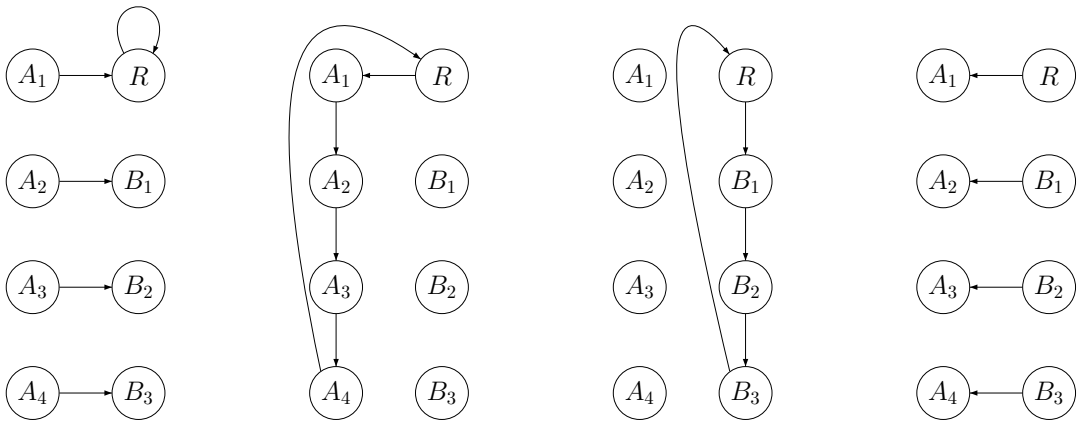
The following operations form the strings s' (the cost of each operation, i.e. the number of letters that form the operation, is also given):

1. From a subset of β_0, the string 1^{h-1} brings the automaton to a configuration that is a subset of β_{h-1}. The coins from R and A_1 are moved to R and from A_i, $i > 1$, to B_{i-1} (Figure 4.13(a)). The cost of this operation is $h-1$.

2. From a subset of β_0, the letter 0 brings the automaton to another subset of β_0. The coins from A_i move to A_{i+1} for $i = 1, \ldots, h-2$. The coin from R goes to A_1 and from A_{h-1} to R (Figure 4.13(b)). The cost of this operation is 1.

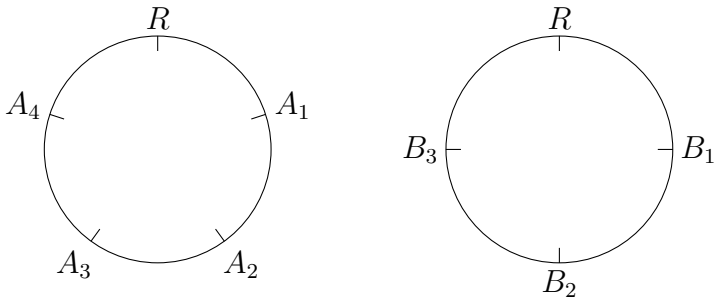
3. From a subset of β_{h-1}, the letter 1 brings the automaton to another subset of β_{h-1}. The coins from B_i move to B_{i+1} for $i = 1, \ldots, h-3$. The coin from R moves to B_1 and from B_{h-2} to R (Figure 4.13(c)). The cost of this operation is 1.

4. From a subset of β_{h-1}, the letter 0 brings the automaton to a subset of β_0. The coins from B_i, $i = 1, \ldots, h-2$, go to A_{i+1}. The coin from R moves to A_1 (Figure 4.13(d)). The cost of this operation is 1.



(a) Operation 1 (b) Operation 2 (c) Operation 3 (d) Operation 4

Figure 4.13: Movements of coins under the operations 1-4.



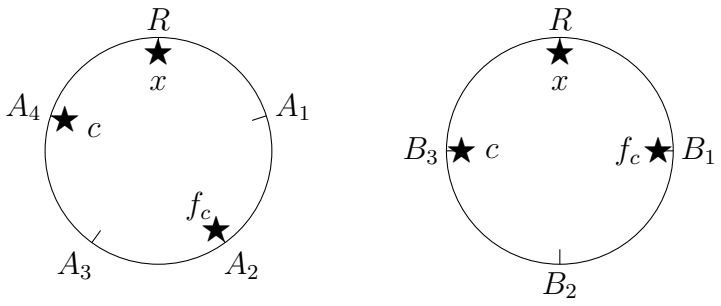
(a) C_1: h positions (b) C_2: $h-1$ positions

Figure 4.14: An automaton equivalent to $\mathcal{F}_5$.

With these operations we may forget about the sets $\beta_1, \ldots, \beta_{h-2}$ and consider only β_0 and β_{h-1}. The sets β_0 and β_{h-1} can be visualized as circles with marked positions $R, A_1, \ldots, A_{h-1}$, for β_0 (let us call it circle C_1), and $R, B_1, \ldots, B_{h-2}$, for β_{h-1} (let us call it circle C_2), as in Figure 4.14. Operations 2 and 3 move all the coins in the clockwise direction on circle C_1 and C_2, respectively. Operation 1 transforms the circle C_1 (Figure 4.14(a)), to the circle C_2 (Figure 4.14(b)). This is done by merging R and A_1 in the new position R and then renaming A_i to B_{i-1} for $0 < i < h$. The operation 4 does the reverse. It first renames B_i to A_{i+1}, then renames R to A_1 and finally inserts a new empty position R between A_{h-1} and A_1.

The initial configuration, L'_h, is the circle C_1 with coins on all positions. Now, the goal is to find a sequence of the operations 1-4 that merges the initial configuration, such that the total cost of the operations is minimal (the lightest merging sequence). It is easy to see that any such sequence moves all the coins to R.

Let the *distance* between two positions, X and Y, of the circle C_i $(i = 1, 2)$, $d(X, Y)$, be the number of hops from X to Y in clockwise direction. The distance



(a) Coins on C_1 (b) Coins on C_2

Figure 4.15: Example of coins on circles C_1 and C_2 for $h = 5$.

between two coins is the distance between their positions. The distance between a coin c and a position X is the distance between the position of c and X, and similarly for the distance between a position and a coin. We assume that $d(X, X) = 0$.

Let the *value* of a coin c, d_c, be the largest distance to any other coin. The coin with the largest distance from c is denoted by f_c. We say that a coin (or a position) is *between* the coins (or positions) A and B if it is on the path from A to B in the clockwise direction, but it is neither on A nor on B.

Example 4.18: In Figure 4.15(a) there are three coins on the circle C_1. We have $d(c, R) = d(c, x) = 1$, $d(c, f_c) = 3$ and f_c is the farthest coin from c. The position R is between c and f_c, but is not between f_c and c (only A_3 is between f_c and c).

For each coin, c, we define a function, P_c, called the *potential* of c. It depends on the position of the coin c and positions of other coins. It also depends on whether the coins are in the circle C_1 or C_2.

The potential of a coin c in a configuration is defined for any configuration with at least two coins by:

$$P_c = P_{0c} + P'_c, \tag{4.31}$$

where:

$$P_{0c} = (d_c - 1)(2h - 1), \tag{4.32}$$

and

$$P'_c = \begin{cases} (i): \ d(c, R) & \text{if the coins are on } C_1, \\ (ii): \ d(c, B_{h-2}) + 2 & \text{if the coins are on } C_2 \\ & \text{and } R \text{ is between } f_c \text{ and } c \text{ or } c \text{ is on } R, \\ (iii): \ d(c, B_{h-2}) + h + 1 & \text{if the coins are on } C_2 \\ & \text{and R } is \text{ between } c \text{ and } f_c \text{ or } f_c \text{ is on } R. \end{cases} \tag{4.33}$$

The definition is valid if there are at least two coins. The value of P'_c is in $\{0 \dots h-1\}$ in case (i), in $\{2 \dots h\}$ in case (ii), and in $\{h+1 \dots 2h-1\}$ in case (iii).

Example 4.19: In Figure 4.15(a) the potentials are the following.

$$\begin{aligned} P_c &= (d(c, f_c) - 1) \cdot (2h - 1) + d(c, R) = 2 \cdot 9 + 1 = 19 \\ P_x &= (d(x, c) - 1) \cdot (2h - 1) + d(x, R) = 3 \cdot 9 + 0 = 27 \\ P_{f_c} &= (d(f_c, x) - 1) \cdot (2h - 1) + d(f_c, R) = 2 \cdot 9 + 3 = 21 \end{aligned}$$

In Figure 4.15(b) the potentials are the following.

$$\begin{aligned} P_c &= (d(c, f_c) - 1) \cdot (2h - 1) + d(c, B_3) + h + 1 \\ &= 1 \cdot 9 + 0 + 5 + 1 = 15 && \text{(case } (iii)\text{)} \\ P_x &= (d(x, c) - 1) \cdot (2h - 1) + d(x, B_3) + 2 \\ &= 2 \cdot 9 + 3 + 2 = 23 && \text{(case } (ii)\text{)} \\ P_{f_c} &= (d(f_c, x) - 1) \cdot (2h - 1) + d(f_c, B_3) + h + 1 \\ &= 2 \cdot 9 + f2 + 5 + 1 = 26 && \text{(case } (iii)\text{)} \end{aligned}$$

Let the *leader* be the coin with the lowest potential (we do not assume that the leader is unique, but we will later see that it is the case). The potential of a configuration is the potential of its leader.

In Figure 4.15(a) the coin c is the leader. The potential of this configuration is equal 19. In Figure 4.15(b) also the coin c is the leader. The potential of this configuration is now equal 15.

Lemma 4.20. *If c has minimal potential of all the coins in a given configuration then it also has minimal value d_c. Moreover, if two coins c_1 and c_2 have the same potential then they have the same value: $d_{c_1} = d_{c_2}$.*

Proof. If the configuration is on C_1 then $0 \leq P'_c \leq h-1$. If the configuration is on C_2 then $2 \leq P'_c \leq 2h-1$. In both cases the range of values of P'_c is less than $2h-1$ and $2h-1$ is the least difference in potential if d_c differs. This proves both statements of the Lemma. □

Lemma 4.21. *Each coin in a fixed configuration has a different potential.*

Proof. Let us assume the contrary, that there are two coins c_1 and c_2 with the same potential. From Lemma 4.20 we know that $d_{c_1} = d_{c_2}$, so $P'_{c_1} = P'_{c_2}$. If the configuration is on C_1 then P'_{c_i} is described by (i) and the coins have to be at the same position on the circle, which contradicts the assumption that they are different. The same happens if both P'_{c_1} and P'_{c_2} are defined by (ii) or both of them are defined by (iii). On the other hand, the ranges of P'_c for cases (ii) and (iii) of (4.33) are disjoint, so no other choice is possible. □

Lemma 4.22. *After each operation 1-4, assuming that there are at least two coins after the operations, the potential of a configuration decreases by at most the cost of the operation. Moreover, in each configuration there is an operation that decreases the potential of the configuration by exactly the cost of the operation, as long as after each operation there are at least two coins.*

Proof. Each possible operation will be considered separately. We will first show that the potential of any coin c decreases by at most the cost of the operation. Then, for any coin c we will find an operation O_c (not necessarily unique) that decreases the potential of c by exactly the cost of O_c. This will prove the lemma, because the potential of the leader c_L may decrease by the cost of O_{c_L}. The leader will also be the leader after the operation O_{c_L} because the potential of no other coin can decrease by a larger value after applying O_{c_L} and the leader had the lowest potential before the operation.

Let us consider the circle C_1 first:

- Operation 1: If d_c does not change then P'_c cannot decrease by more than $h-1$. Otherwise, d_c decreases by exactly one. We have to show that in this case P'_c increases by at least h. Before the operation we had $P'_c = d(c, R)$. As the distance between c and f_c decreased after the operation, there are two cases.

  The first one is that c is on R before the operation. Then it stays on R after the operation as well. The new value of P'_c is described by (ii) of (4.33), which means that it increased from 0 to $d(R, B_{h-2}) + 2 = h$.

  The second case is that c is not on R before the operation. As the distance, d_c, decreased, f_c must have been between R and c. Thus, after the operation f_c is between R and c or f_c is on R. The new value of the potential is described by (iii) of (4.33): $P'_c = d(c, B_{h-2}) + h + 1$. Since c is not on R, $d(c, B_{h-2}) = d(c, R) - 1$ and P'_c increases by exactly h.

- Operation 2: The potential of a coin c either decreases by one, if c is not on R, or increases by $h-1$, if c is on R. Thus, in both cases the potential decreases by at most one.

If the coin c is not on R, operation 2 decreases the potential of c by 1. Otherwise, i.e. if c is on R, operation 1 decreases the potential of c by $h-1$.

For the circle C_2:

- Operation 3: If P'_c before and after the operation are both described by the same part (ii) or (iii) of (4.33) then the potential decreases by at most one. If before the operation the potential is described by (ii) and afterwards by (iii), it can only increase (note that $h \geq 3$). Finally, if the potential before the operation is described by (iii) and afterwards by (ii) then at start c is on B_{h-2} and afterwards c is on R. Then the difference in the potential is:

$$\begin{aligned} \{P'_c\}_{\text{after}} - \{P'_c\}_{\text{before}} &= d(R, B_{h-2}) + 2 - d(B_{h-2}, B_{h-2}) - h - 1 \\ &= h - 2 + 2 - 0 - h - 1 = -1, \end{aligned} \tag{4.34}$$

  which is correct.

- Operation 4: If d_c increases after applying the operation then P_c increases as well. Otherwise initially c is on R or R is between f_c and c. In both cases P'_c is described by part (ii) of (4.33). At start $P'_c = d(c, B_{h-2}) + 2$. After the

operation: $P'_c = d(c, R)$ (case (i) of (4.33)). As c cannot be on R after the operation, $\{d(c, B_{h-2})\}_{\text{before}} = \{d(c, R)\}_{\text{after}} - 1$, and

$$\{P'_c\}_{\text{after}} - \{P'_c\}_{\text{before}} = \{d(c, R)\}_{\text{after}} - \{d(c, B_{h-2})\}_{\text{before}} - 2 = -1, \quad (4.35)$$

thus the potential decreased by exactly one.

Operation 4 decreases the potential of c by one if the potential is described by the part (ii) of the equation. Otherwise, operation 3 decreases the potential of c by 1.

Note that in many cases the operation mentioned in the lemma is not unique. □

Proof of Theorem 4.17. The length of the shortest merging sequence s for the set L'_h of states of $\mathcal{F}_h$ is the cost of the lightest merging sequence s_o of operations 1-4 for L'_h. The sequence s can be obtained from s_o by substituting the operations by their corresponding string. We have already stated that such a sequence s also merges the set L_h of states of D_h.

Operation 1 is the only one that reduces the number of coins. So the last operation of s_o is operation 1, that merges the coins in R and A_1. The cost of this final operation is $h - 1$. Let us call the configuration with coins just in R and A_1 the configuration C.

By Lemma 4.22 it is always possible to decrease the potential of a configuration by the cost of an operation as long as the final configuration has at least two coins. The only configuration that may lead from a configuration with at least two coins to a configuration with just one coin is the configuration C and C has the lowest potential of all configurations with at least two coins. It is then always possible to reach the configuration C starting from any configuration with at least two coins and using only optimal operations, i.e. the operations that have the cost equal to the drop in the potential.

The potential of the initial configuration is

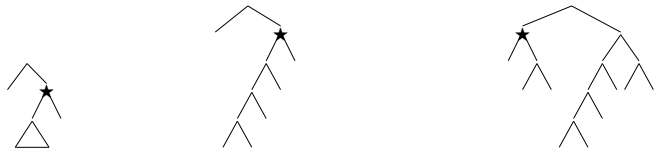
$$P_0 = 0 + (h - 2)(2h - 1) = 2h^2 - 5h + 2. \quad (4.36)$$

The potential of the configuration C is $P_1 = 0$. There is a sequence s_o of operations 1-4 of cost $P_0 - P_1$ that leads from the initial configuration to the configuration C. No sequence leading from the initial configuration to the configuration C can have lower cost. Thus, the length of the shortest merging string s for the set L'_h of $\mathcal{F}_h$ is

$$|s| = P_0 - P_1 + h - 1 = 2h^2 - 4h + 1. \quad (4.37)$$

The sequence s is also a synchronizing sequence for the automaton $\mathcal{F}_h$. Indeed, any shortest merging string for the set L'_h has to start with 1^{h-1}. The application of 1^{h-1} to either the set of all states of $\mathcal{F}_h$ or to the set L'_h results in the same configuration: the coins on the states R, B_1, ..., B_{h-2}.

Finally, by Theorem 4.7, the synchronizing string for $\mathcal{F}_h$ we found is also a synchronizing string for D_h. D_h cannot have any shorter synchronizing string, because it would also be a merging string for the set L'_h of states of $\mathcal{F}_h$, which is impossible. □



(a) A tree G_k (b) G_4, $s = 0^4$ (c) C_4, $s = 1110^4$ or $s = 1010^4$

Figure 4.16: The nodes with the longest merging string for the two families C_k and G_k.

The minimized Huffman automaton for D_h has $K = 2(h-1)$ nodes. Even though it contains a letter that reduces the number of coins by $h - 2 = \frac{K}{2} - 1$ (a letter of deficiency $\frac{K}{2} - 1$), its shortest synchronizing string is of length $2h^2 - 4h + 1 = \frac{K^2}{2} - 1$, which is quadratic in K. This makes the automata D_h interesting in themselves.

The results of the search allow us to state the following conjecture.

Conjecture 4.23. The length of the shortest synchronizing string s for a code with N codewords, $N \geq 9$, with h being the length of the longest codeword, is at most:

$$|s| \leq \min(2N - a, 2h^2 - 4h + 1), \tag{4.38}$$

where a is 7 for odd N and 9 for even N.

4.3.2 Long merging string

For trees of fixed size, N, the length of the shortest merging string for $\{n, \varepsilon\}$ in the worst case is equal $N-2$, for $N = 3, \ldots, 20$, apart from $N = 6$. For $N = 6$ the worst-case length is equal $N - 1 = 5$. Two families of trees have the worst-case shortest merging strings. The first one corresponds to the code

$$G_k = \{0, 10^k\} \cup \{10^i 1 | i < k\}, \quad k \geq 1, \tag{4.39}$$

and gives the worst-case trees for N from 3 to 20, apart from $N = 6$. The size of the tree G_k is $N = k + 2$. The merging string for the set $\{1, \varepsilon\}$ is of length $N - 2$. The structure of these trees is shown in Figure 4.16(a). The tree G_4 is shown in Figure 4.16(b). These figures also indicate the node whose merging string with ε is the longest.

The other family of trees is the family C_k (see (4.25) and Figure 4.7(a)) with even k (odd number of codewords). The merging string for the set $\{0, \varepsilon\}$ is of length $N-2$ and this is the worst case for $N = 7, 9, 11, \ldots 19$. The node with the longest merging string with ε is 0 (Figure 4.16(c)).

There are also additional worst-case trees for $N = 5, 6, 7, 9, 10, 12$. They correspond to neither the trees C_k nor G_k. They are presented in Figure 4.17. The nodes with the longest merging string with the root are marked with a star.

The worst-case trees among trees of fixed height, h, are the trees D_h (Equation (4.26) and Figure 4.10).

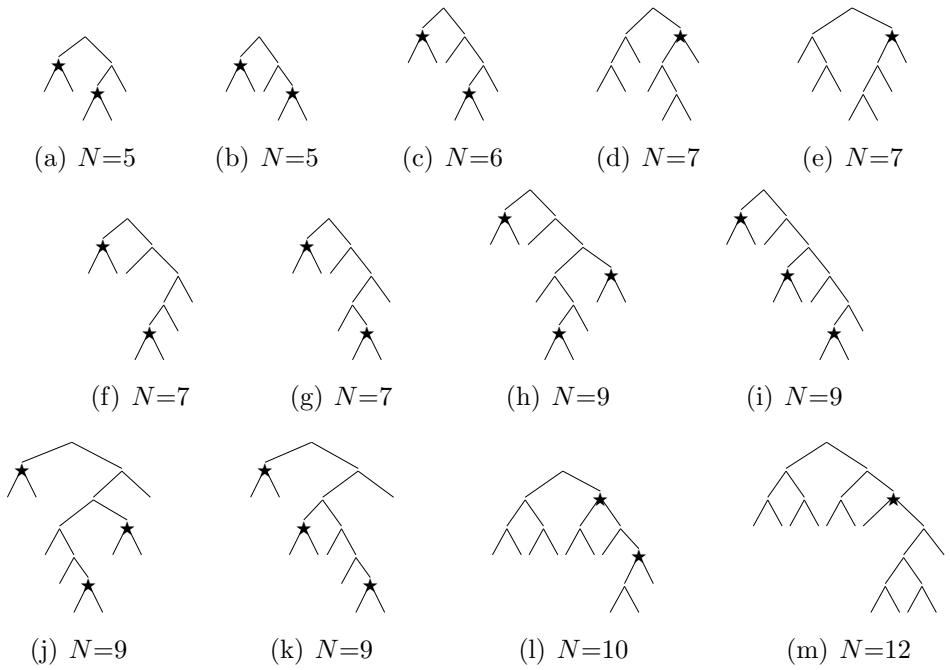


Figure 4.17: Worst case trees in terms the length of the shortest merging string. The nodes that require a merging string of the worst-case length are marked with a star. The trees of the families C_k and G_k are omitted.

Theorem 4.24. *The upper bound on the length of the shortest merging string for any pair $\{n, \varepsilon\}$, where n is a state of D_h, is $\lceil h^2 - \frac{3}{2}h \rceil$. For odd h it is achieved by the pair $\{0^{(h-1)/2}, \varepsilon\}$. For even h it is achieved by pairs $\{x, \varepsilon\}$, where x is any binary string of length $\frac{h}{2}$ containing at least one 0.*

Proof. The length of the shortest merging string is given by the potential of the position plus $h - 1$ (see the proof of Theorem 4.17). It is enough to find a two-coin configuration with one coin on R that has the largest potential.

For even h, the maximum of the minimal distance between the two coins is $h/2$ and it is possible only in the circle C_1. The second coin is then on $A_{h/2}$. In this configuration the coin with minimal potential is the one on R. The length of the shortest merging string is:

$$\begin{aligned} |s| &= P_R + (h-1) = P_{0R} + P'_R + (h-1) \\ &= (d_R - 1)(2h-1) + d(R,R) + (h-1) \\ &= (\tfrac{h}{2} - 1)(2h-1) + 0 + (h-1) \\ &= h^2 - \tfrac{3}{2}h. \end{aligned}$$

For odd h, the maximum of the minimal distance between coins is $(h-1)/2$. Such a configuration is possible on either the circle C_1 and C_2. Let us assume for a moment that the coins are in C_2. The second coin must be $B_{(h-1)/2}$ to get the distance $(h-1)/2$ between the coins. The coin with minimal potential is the one on

R. The length of the shortest merging string is:

$$\begin{aligned}|s| &= P_R + (h-1) = P_{0R} + P'_R + (h-1)\\ &= (d_R - 1)(2h-1) + d(R, B_{h-2}) + 2\\ &= \left(\tfrac{h-1}{2} - 1\right)(2h-1) + (h-2) + 2 + (h-1)\\ &= h^2 - \tfrac{3}{2}h + \tfrac{1}{2}\end{aligned}$$

No potential of any position on C_1 can be higher, because the value of P'_c on the circle C_1 never exceeds $h-1$ (it is equal h in the position on C_2 analyzed above). □

The results of the search allow us to state the following conjecture.

Conjecture 4.25. For any Huffman automaton, $\mathcal{T}$, corresponding to a code with N codewords, with h being the length of the longest codeword, the length of the shortest merging string, s_n, for a set $\{n, \varepsilon\}$, where n is any state of $\mathcal{T}$ is bounded by:

$$|s| \leq \min(N-2, \lceil h^2 - \tfrac{3}{2}h \rceil), \tag{4.40}$$

if $N \neq 6$, and $|s| \leq 5$ for $N = 6$.

4.4 Synchronizing codewords

This section presents two algorithms for finding all synchronizing codewords in a Huffman code. Both algorithms work in the time proportional to the sum of codewords' lengths. The first one is very simple and also efficient. The other one is more involved but, additionally, after preprocessing in $O(N)$, it is able to answer in $O(|w|)$ if a codeword w is synchronizing.

4.4.1 Simple algorithm

To check if a particular codeword w is synchronizing one can traverse the Huffman tree from each node using the string w and check whether the final node is always the root. Direct implementation of this method requires $O(N|w|)$ operations for each codeword w. This gives the total $O(N \sum |w_i|)$ operations to find all synchronizing codewords in a code.

The time complexity can be improved by changing the order of the computations. We can first chose a node n of the Huffman tree and then check which codewords synchronize n. The codewords form a complete binary tree, so they share many common prefixes. Checking which codewords synchronize the node n can be done in one DFS traversal of the tree of codewords. The number of operations for each node is only $O(N)$, so $O(N^2)$ operations suffice to find all synchronizing codewords. In the worst case, i.e. when $\sum |w_i| = O(N^2)$, this is by factor N better than before.

Algorithm 4.1 is based on the idea just described with an additional optimization, to be explained later. The call SEARCHSYNC(ε, v) in the body of the algorithm marks the nodes that do not synchronize the vertex v. The first parameter

Algorithm 4.1: Simple Algorithm.

Input: T — a Huffman tree.
Output: For each node v, the field v.SYNC set to TRUE iff $\pi(v)$ is a synchronizing codeword.

```
1  procedure searchSync(x, y)
2    if x is a leaf then
3      if y is not a leaf then x.SYNC ← FALSE;
4    else
5      if y is a leaf then
6        if not x.PROCESSED then
7          x.PROCESSED ← TRUE;
8          y ← ε;
9        else return;
10     searchSync(x.LEFT, y.LEFT);
11     searchSync(x.RIGHT, y.RIGHT);
12 end procedure
13 forall node v of T do
14   v.SYNC ← TRUE;
15   v.PROCESSED ← FALSE;
16 forall node v of T do
17   searchSync(ε, v);          ◁ Check which codewords do not synchronize v
```

in searchSync(x, y), x, is the node that determines the prefix $\pi(x)$ of a codeword being analyzed. The second parameter, y, is the result of traversing the tree from the initial node v using the word $\pi(x)$. If the node x is a leaf then the algorithm checks whether the codeword $\pi(x)$ synchronized the node v. This holds if and only if the node y is also a leaf.

Algorithm 4.1 uses an optimization that prunes computations that have already been done during algorithm's execution. Lines 6, 7 and 9 prune the calls to searchSync for pairs $(x, y = \varepsilon)$ that have been processed before. With this optimization the number of operations is proportional to S_T — the sum of the sizes of all the subtrees of the code tree T, which is $O(\sum |w_i|)$ by Lemma 4.2.

The properties of the algorithm can be summarized as follows.

Theorem 4.26. *After the execution of Algorithm 4.1 on a Huffman tree T, the field l.SYNC for a leaf $l \in T$ is* TRUE *if and only if $\pi(l)$ is a synchronizing codeword for the Huffman code $\mathcal{C}(T)$. The time complexity for Algorithm 4.1 is $O(\sum_i |w_i|)$, where w_i are codewords of the code $\mathcal{C}(T)$.*

4.4.2 Improved algorithm

The new method (Algorithms 4.2 and 4.3) requires $O(N)$ operations for preprocessing and then it is able to answer in $O(|w|)$ time whether a codeword w is synchronizing.

For the problem of finding all the synchronizing codewords this does not give a better complexity than Algorithm 4.1. Nevertheless, the new method can be used to check in $O(N)$ time if a particular codeword is synchronizing. It is not possible with Algorithm 4.1. The method is based on the following theorem:

Theorem 4.27. *A codeword w is synchronizing if and only if the two conditions are met:*

1. *There is no codeword w' in the code such that $w' = awb$, where both a and b are nonempty words (i.e. w is a subword of w' that is not a prefix nor a suffix; w will be called here a* proper *subword of w').*
2. *Let $w = w''w'$ be such a word that there is a node v of the Huffman tree such that $\delta^*(v, w'') = \varepsilon$ and for no proper prefix p of w'' $\delta^*(v, p) = \varepsilon$. Then w' is a string of codewords.*

Proof. It is easy to see that for a synchronizing codeword both conditions are met. To see the reverse, let us take an internal node $v \neq \varepsilon$ and let us traverse the tree from v using w. We cannot finish before reaching a leaf because that would contradict 1, so we end up in a leaf after a prefix w'', with $w = w''w'$. Then, by the second condition, w' is a string of codewords and the further traversal ends in the root. It follows that w synchronizes any vertex v. □

For each node v of the Huffman tree the following values will be used in the algorithms:

- v.BORD — the lowest node v' such that $\pi(v')$ is a proper suffix of $\pi(v)$,
- v.LEAF — the lowest leaf node v' such that $\pi(v')$ is a proper suffix of $\pi(v)$,
- v.NONLEAF — the lowest non-leaf node v' such that $\pi(v')$ is a proper suffix of $\pi(v)$,
- v.DEPTH — the distance from the root (we have ε.DEPTH $= 0$).

By the lowest node it is meant the node with the largest DEPTH. If no node fulfilling the properties exists, the value is NULL. These pieces of information can be computed with $O(N)$ operations (see Algorithm 3.8).

The method uses two algorithms. Algorithm 4.2 finds all codewords that do not satisfy statement 1 of Theorem 4.27, i.e. the codewords that are proper subwords of other codewords. It marks the corresponding leaf nodes, x, with x.SUBWORD $\leftarrow$ TRUE. This is done in lines 5 and 6. These lines are based on the observation that the field v.LEAF for non-leave nodes v gives a codeword w that is a proper subword of another codeword w'. Any suffix of such a proper subword is also a proper subword, so the target of the field v.LEAF for such a codeword is set as SUBWORD too.

Algorithm 4.2 also computes all the possible prefixes w'' defined in Condition 2 of Theorem 4.27. Line 8 sets the flag v.INLEAF for each vertex v such that $\pi(v)$ is such a prefix w''. These values will be used in Algorithm 4.3. The complexity of Algorithm 4.2 is $O(N)$.

Algorithm 4.2: Computing SUBWORD and preprocessing for Algorithm 4.3.

Input: T — a Huffman tree.
Output: The fields SUBWORD and INLEAF for each node of the tree.

```
1 forall node v of T do
2     v.SUBWORD ← FALSE;
3     v.INLEAF ← FALSE;
4 forall node v of T in reversed BFS order do
5     if v is not a leaf or v.SUBWORD = TRUE then
6         if v.LEAF is not NULL then v.LEAF.SUBWORD ← TRUE;
7     if v is a leaf or v.INLEAF then
8         if v.BORD is not NULL then v.BORD.INLEAF ← TRUE;
```

The second algorithm of the method, Algorithm 4.3, answers in $O(|w|)$ whether a codeword w is synchronizing. It uses the information x.SUBWORD and x.INLEAF computed by Algorithm 4.2. First, the codeword is rejected if it is a proper subword of another codeword. Otherwise, Algorithm 4.3 processes the word w bit by bit and analyzes the decoders of w that start at each marked suffix of w, i.e. at the nodes v with v.INLEAF set to TRUE. The array T keeps track of which decoders are *active*. A decoder becomes active when it starts at some position k_1. Then the value $T[k_1]$ is set to TRUE (line 14). The decoder becomes inactive when it reaches a leaf at some position k_2. Then $T[k_1]$ is set to FALSE (line 11). But each decoder that reaches a leaf reappears in the root (see Chapter 2). This is equivalent to activating some other decoder at $T[k_2]$, which is done in line 12.

It can be shown that the states of decoders that may have appeared in a leaf can be found on the list $\{v.\text{LEAF}, v.\text{LEAF.LEAF}, \ldots\}$ (line 9; see Section 3.5.2 for details). The field x.DEPTH for such a state gives the number of bits processed by the decoder. The bit number where the decoder started is $k_1 = i - x.\text{DEPTH}$, where i is the number of the current bit of w. The number k_1 is the index to the array T, where it can be checked whether the decoder is active and, if so, deactivate it.

At the end, Algorithm 4.3 checks if there is an active decoder different from the one that has just started. If so, the codeword w is not synchronizing, because the sufix processed by this decoder is not a sequence of codewords and condition 1 from Theorem 4.27 does not hold.

Theorem 4.28. *Algorithm 4.3 uses $O(|w|)$ operations.*

Proof. It is enough to prove that each element of the array T is accessed in line 10 at most once. Suppose the contrary, an element k of T is accessed twice, for $i = i_1$ and for $i = i_2$, $i_1 < i_2$. Then, there are two leaves, x_1 and x_2, with $k = i_1 - x_1.\text{DEPTH}$ and $k = i_2 - x_2.\text{DEPTH}$. The strings $\pi(x_1)$ and $\pi(x_2)$ are codewords and they are suffixes of $w[0..i_1)$ and $w[0..i_2)$ of length $i_1 - k$ and $i_2 - k$, respectively. Thus, the position of w_l in w, $l = 1, 2$, is $i_l - (i_l - k) = k$. This means that w_1 is a prefix of w_2, which is a contradiction since the code is prefix-free. □

Algorithm 4.3: Checking in $O(|w|)$ if a codeword w is synchronizing.

Input: T — a Huffman tree, w — a codeword of the corresponding Huffman code.

Output: w.SYNC set to true iff w is a synchronizing codeword.

```
1  if v.SUBWORD = TRUE then
2      w.SYNC ← FALSE;
3  else
4      w.SYNC ← TRUE;
5      v ← ε;                                  ◁ current node on the path corresponding to w
6      for i ← 1 to |w| do
7          v ← δ_T(v, w[i − 1]);                                   ◁ v = δ*_T(ε, w[0..i))
8          T[i] ←FALSE;
9          forall x in (v.LEAF, v.LEAF.LEAF, ...) do
10             if T[i − x.DEPTH] = TRUE then
11                 T[i − x.DEPTH] ← FALSE;          ◁ the decoder reached a leaf
12                 T[i] ← TRUE;                     ◁ and restarts from the root
13         if v.INLEAF = TRUE then
14             T[i] ← TRUE                          ◁ a new decoder starts here
15     if some T[i], i = 1 ... (|w| − 1) is TRUE then
16         w.SYNC ← FALSE;
```

If processing all the codewords is needed, Algorithm 4.3 may be improved. Instead of analyzing each codeword separately, it may analyze all the codewords at the same time, reusing the computations for common prefixes. This optimization decreases the overall number of operations, but unfortunately it does not decrease the worst-case complexity, because the time needed for Algorithm 4.3 for processing each bit is not constant. Nevertheless, experiments have shown that with this trick the number of operations is reduced by about three times for the tested trees.

It should be noted that both algorithms for finding all the synchronizing codewords for a code are practical. They give instantly the results even for codes with thousands of codewords. They were tested, in particular, on the extended Huffman code for Jpeg compression, defined in Section 3.10, which contains about 32 thousand codewords. On the other hand, the methods that use the synchronization graph, presented in Section 3.3, are impractical for such codes.

4.5 Conclusions

In this chapter we analyzed the relation between Huffman codes and a certain class of finite automata. An upper bound of the shortest merging string for a pair of states of these automata was shown. This result was then used to prove an upper bound on the length of the shortest synchronizing string for any synchronizing Huffman code.

The proof was constructive and an algorithm that constructs a synchronizing string achieving the bound was given. The algorithm is faster than the general algorithm of Eppstein [18], the best known at the moment. It also constructs shorter synchronizing strings. Also an algorithm for testing whether a Huffman code is synchronizing was presented. This algorithm is also faster than the one of Eppstein [18].

We tested the lengths of the shortest merging string and of the shortest synchronizing string on all codes of size from 3 to 20 and on all codes with the length of the longest codeword from 2 to 5. Three classes of worst-case codes were found. Two of them corresponded to the worst-case length of a synchronizing string. For these codes, the exact length of the shortest synchronizing strings was computed. This allowed us to formulate conjectures that these codes are the worst for any code size.

It is interesting that the length of the shortest synchronizing strings for the worst-case codes is much lower than the bound proven. Improvement of the bounds remains an open problem.

Finally, two efficient algorithms for finding all synchronizing codewords of a Huffman code were presented. The algorithms are fast and simple to implement, so they are of practical interest. They may be applied for searching for codes with the best synchronization properties, expressed in the number of synchronizing codewords and their lengths. Low complexity of the algorithms increase the number of codes that can be analyzed efficiently.

Appendix A

Proof of Theorem 3.47

The theorem will be proved in several steps. Let us denote:

$$w = a_0 a_1 \dots a_{n-1}, \tag{A.1}$$

where $a_i \in \Sigma$. Thus, $|w| = n$.

Let us define $P_w \subseteq \{0, \dots, n-1\}$ as

$$k \in P_w \iff a_k a_{k+1} \dots a_{n-1} = a_0 a_1 \dots a_{n-k-1}. \tag{A.2}$$

The set P_w corresponds to the set of suffixes of w that are its prefixes.

Lemma A.1. *P_w has the following properties, for any $k, l \in P_w$, $k \leq l$, and for any $i \geq 0$:*

$$k + l < n \Rightarrow k + l \in P_w, \tag{A.3}$$

$$i \cdot k < n \Rightarrow i \cdot k \in P_w, \tag{A.4}$$

$$k + (l-k)i < n \Rightarrow k + (l-k)i \in P_w. \tag{A.5}$$

Proof. For (A.3) we have from (A.1) and (A.2) for $(k+l) < n$:

$$\begin{aligned} a_{k+l} a_{k+l+1} \dots a_{n-1} &= a_l a_{l+1} \dots a_{n-k-1} \\ &= a_0 a_1 \dots a_{n-k-l-1}, \end{aligned} \tag{A.6}$$

so $(k+l) \in P_w$.

Equation (A.4) can be proved by induction. It is true for $i = 0$ and $i = 1$. Then, assuming that (A.4) is true for $i-1$ and 1, we have $(i-1)k \in P_w$ and $k \in P_w$ thus from (A.3) also $i \cdot k \in P_w$ if only $i \cdot k < n$.

Finally, for Equation (A.5), let $v = a_k \dots a_{n-1}$. Then $(l-k) \in P_v$. From (A.4), $(l-k)i \in P_v$ for $i \geq 0$ such that $(l-k)i < n-k$. It follows that $(l-k)i + k \in P_w$ for the same values of i. □

Definition A.2. $m \in P_w$ is a *secondary* appearance in P_w if it follows from the property (A.5) of P_w, i.e.

$$\exists_{k,l \in P_w, 0 \leq k < l} \exists_{i \geq 2} (m = k + (l-k)i) \tag{A.7}$$

The number l in (A.7) will be called a *parent* of the appearance p; p will be called a *child* of the appearance l.

Note that an appearance may have zero or more children and a secondary appearance has one or more parents.

Definition A.3. $m \in P_w$ is a *primary* appearance in P_w if it is not a secondary appearance of $m \in P_w$.

Primary appearances are the elements that do not have a parent.

Definition A.4. An *interval*, $D(p)$, of a primary appearance $p \in P_w$, is the smallest $x > 0$ such that for all $i > 0$, $(p + x \cdot i) \in P_w$ if only $(p + x \cdot i) < n$. In particular x may be such that $p + x > n$.

Lemma A.5. *The number of primary appearances in P_w is less or equal $\lfloor \log n \rfloor + 1$.*

Proof. Let p_1, p_2, ... be all the primary appearances in P_w in increasing order. We will show that their distances decrease exponentially.

The appearance $p_1 = 0$ corresponds to the whole word w and $D(p_1) \leq n$. For any p_k we have $(p_k + iD(p_k)) \in P_w$ for $i \geq 0$. It follows that for some $i \geq 0$:

$$p_k + iD(p_k) \leq p_{k+1} < p_k + (i+1)D(p_k). \tag{A.8}$$

Then either

$$p_{k+1} - (p_k + iD(p_k)) \leq D(p_k)/2 \tag{A.9}$$

or

$$(p_k + (i+1)D(p_k)) - p_{k+1} \leq D(p_k)/2 \tag{A.10}$$

(otherwise after adding (A.9) and (A.10) we get $D(p_k) > D(p_k)$). Hence, from (A.5) we get $D(p_{k+1}) \leq D(p_k)/2$. From $D(p_k) \geq 1$ follows the Lemma. □

In the proof of Theorem 3.47 we will add new bits at the end of w to remove as many primary appearances as possible. We will also analyze how removing a primary appearance influences secondary appearances by looking at the parent-child relations.

Definition A.6. The *blocking bit* for $k \in P_w$ is a letter $B(k)$ such that $k \notin P_{wB(k)}$.

It is easy to see that $B(k)$ is $\overline{a_{n-k}}$, the negation of the bit a_{n-k}.

Lemma A.7. *If b is a blocking bit for an appearance l then it is a blocking bit for all its descendants (the children of l, the children of the children of l, etc.).*

Proof. It is enough to prove that b is a blocking bit for any child of l. The proof for the other descendants follows by induction.

Let $m \in P_w$ be a secondary appearance and let l be its parent. Then we have some $k \in P_w$ and $i \geq 2$ such that $m = k + (l-k)i$. If b is a blocking bit for l then $b = \overline{a_{n-l}}$. However, $(l-k)$ is a period of the word $a_0 a_1 \ldots a_{n-k-1}$, i.e. $a_{j+(l-k)} = a_j$ for any $j \geq 0$ such that $j + (l-k) < n-k$, i.e. $j < n-l$ (it follows from $a_j = a_{l+j} = a_{k+j}$ for $l + j < n$ as $k, l \in P_w$). Then

$$\begin{aligned} a_{n-m} &= a_{n-k-(l-k)i} = a_{n-k-(l-k)(i-1)} \\ &= \ldots = a_{n-k-(l-k)\cdot 1} = a_{n-l}, \end{aligned} \tag{A.11}$$

so b is also the blocking bit for m. □

Note that in the proof of Lemma A.7 the blocking bit for k may not necessarily be the blocking bit for m, because the chain of equalities (A.11) cannot be further extended to $a_{n-l} = a_{n-l+(l-k)} = a_{n-k}$, since the condition $j < n - l$ is not fulfilled (here $j = n - l$ and $j + (l - k) = n - k$).

By Lemma A.7, it is enough to focus on removing the appearances that have no parent, which are the primary appearances.

Lemma A.8. *Let $Q \subseteq P_w$, $0 \notin Q$. There exist a word v, $|v| \leq \lfloor \log |Q| \rfloor + 1$, such that $P_{wv} \cap Q = \emptyset$.*

Proof. We will construct the words $v_0 = \epsilon$, v_1, v_2, ..., $v_m = v$, $|v_i| = i$ and $v_{i+1} = v_i b_i$, where b_i is a single bit. Let $Q_i = P_{wv_i} \cap Q$. We have $Q_0 = Q$. The bit b_i will be chosen in the following way. The elements of Q_i will be divided into two groups, depending on their blocking bit in P_{wv_i}. Then, b_i will be the blocking bit that corresponds to the larger group. We always have $|Q_{i+1}| \leq |Q_i|/2$. So after $m \leq \lfloor \log |Q| \rfloor + 1$ steps we get $Q(m) = \emptyset$. We will prove that v_m is the word v from the Lemma. Indeed, let us assume that $k \in Q$ was in the chosen group that determined the bit b_i. Then the $(n + i)$-th bit in wv is b_i and the $(n - k + i)$-th bit in wv is $\overline{b_i}$. Thus $k \notin P_{wv}$. □

Proof of Theorem 3.47. Let us take Q as the set of all primary appearances of P_w other than 0. From Lemma A.5: $|Q| \leq \lfloor \log n \rfloor$. Thus, applying Lemma A.8 to Q we get a word v with $|v| \leq \lfloor \log \log n \rfloor + 1$ such that $P_{wv} \cap Q = \emptyset$. We will show that v is the word whose existence is stated in the theorem.

Indeed, if we had $wv = xp$ and $|x| = k$ with $0 < k < n$ then $k \in P_{wv}$ and also $k \in P_w$ (see (A.2)). But k cannot be a primary appearance in P_w, because otherwise $k \in Q$ and then $k \notin P_{wv}$. By Lemma A.7, it cannot be a secondary appearances in P_w either because by blocking the primary appearances their descendants were blocked as well, so $k \notin P_{wv}$. □

In Theorem 3.47 one cannot assume that it is possible to find such a short suffix to achieve $|x| = 0$ (i.e. no $|x| \geq n$ is allowed either). Indeed, the counterexample is:

$$w = 0 \underbrace{11 \ldots 1}_{n-2} 0. \tag{A.12}$$

The shortest extension w' of w such that no proper prefix of w' is its suffix is:

$$w' = 0 \underbrace{11 \ldots 1}_{n-2} 0 \underbrace{11 \ldots 1}_{n-1} \tag{A.13}$$

(the last letter must be a 1; then to eliminate $n - 1$ from $P_{w'}$, w' needs to have at least $n - 1$ ones after the last zero).

The theorem is true also for any larger alphabet Σ, but the result can be improved. For instance, in Lemma A.8 we would have $v \leq \lfloor \log_{|\Sigma|} |Q| \rfloor + 1$.

The construction given in the proof can be used as an algorithm for finding such extensions of words. However, one may also try all possible suffixes of length $\leq \lfloor \log \log n \rfloor + 1$, as for $n < 2^{16}$ the length of such a suffix is at most 4.

Bibliography

[1] D. S. Ananichev and M. V. Volkov. Synchronizing generalized monotonic automata. *Theor. Comput. Sci.*, 330(1):3–13, 2005.

[2] D. S. Ananichev, M. V. Volkov, and Yu. I. Zaks. Synchronizing automata with a letter of deficiency 2. *Theor. Comput. Sci.*, 376(1-2):30–41, 2007.

[3] A. Apostolico and A. S. Fraenkel. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Trans. Inf. Theor.*, 33(2):238–245, 1987.

[4] Marie-Pierre Béal. A note on Černý conjecture and rational series. Preprint Institute Gaspard-Monge, Université de Marne-la-Vallé, January" 2003.

[5] Jean Berstel and Dominique Perrin. *Theory of Codes*. Academic Press, Inc., Orlando, FL, USA, 1985.

[6] Marek Tomasz Biskup. Guaranteed synchronization of Huffman codes. In *Proc. 18th IEEE Data Compression Conference (DCC'08)*, pages 462–471, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[7] Marek Tomasz Biskup. Shortest synchronizing strings for huffman codes. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *MFCS*, volume 5162 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2008.

[8] Marek Tomasz Biskup. A word that does not appear in encoded message as a resynchronization marker. In *Proceedings of the IEEE Information Theory Workshop*. Porto, Portugal, 2008.

[9] A. Bookstein and S. Klein. Is Huffman coding dead. *Computing*, 50(4):279–296, December 1993.

[10] N. Brisaboa, A. Fariña, G. Navarro, and M. Esteller. (s,c)-dense coding: An optimized compression code for natural language text databases. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2857, pages 122–136. Springer, 2003.

[11] N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In *Proc. 25th European Conference on Information Retrieval Research (ECIR)*, LNCS 2633, pages 468–481, 2003.

[12] Renato M. Capocelli, Luisa Gargano, and Ugo Vaccaro. On the characterization of statistically synchronizable variable-length codes. *IEEE Trans. Inform. Theory*, 34(4):817–825, July 1988.

[13] Renato M. Capocelli, Alfredo De Santis, Luisa Gargano, and Ugo Vaccaro. On the construction of statistically synchronizable codes. *IEEE Trans. Inform. Theory*, 38(2):407–414, March 1992.

[14] Ján Černý. Poznámka k. homogénnym experimentom s konecnými automatmi. *Mat. fyz. čas SAV*, 14:208–215, 1964.

[15] M. Crochemore and Wojciech Rytter. *Text algorithms*. Oxford University Press, 1994.

[16] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.*, 18(2):113–139, 2000.

[17] S. Deorowicz. *Universal lossless data compression algorithms*. PhD dissertation, Silesian University of Technology, 2003.

[18] David Eppstein. Reset sequences for monotonic automata. *SIAM J. Comput.*, 19(3):500–510, 1990.

[19] Adrian Escott and Stephanie Perkins. Binary Huffman equivalent codes with a short synchronizing codeword. *IEEE Trans. Inform. Theory*, 44(1):346–351, January 1998.

[20] Shimon Even. Test for synchronizability of finite automata and variable length codes. *IEEE Trans. Inform. Theory*, 10:185–189, July 1964.

[21] Yong Fang and Jechang Jeong. VLR-based optimal positioning of resynchronization markers. In *DCC*, page 381. IEEE Computer Society, 2007.

[22] Thomas J. Ferguson and J. H. Rabinowitz. Self-synchronizing Huffman codes. *IEEE Trans. Inform. Theory*, 30(4):687–693, July 1984.

[23] A.S. Fraenkel and S.T. Klein. Bidirectional Huffman coding. *The Computing Journal*, 33(4):296–307, 1990.

[24] Christopher F. Freiling, Douglas S. Jungreis, François Théberge, and Kenneth Zeger. Almost all complete binary prefix codes have a self-synchronizing string. *IEEE Trans. Inform. Theory*, 49(9):2219–2225, September 2003.

[25] E. N. Gilbert. Synchronization of binary messages. *IEEE Trans. Inform. Theory*, 6:470–477, September 1960.

[26] Solomon W. Golomb. Run-length encodings. *IEEE Trans. Inform. Theory*, 12:399–401, July 1966.

[27] L. J. Guibas and A. M. Odlyzko. Maximal prefix-synchronized codes. *SIAM J. Appl. Math.*, 35(2), September 1978.

[28] S. Hemami. Robust image transmission using resynchronizing variable-length codes and error concealment. *IEEE Journal of Selected Areas in Communications*, June 2000.

[29] Yuh-Ming Huang and Sheng-Chi Wu. Shortest synchronizing codewords of a binary Huffman equivalent code. In *ITCC '03: Proceedings of the International Conference on Information Technology: Computers and Communications*, page 226, Washington, DC, USA, 2003. IEEE Computer Society.

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

[31] Li-Wei; Kang and Jin-Jang Leou. A survey of error resilient coding schemes for image and video transmission based on data embedding. In *Proceedings of the IEEE Asia-Pacific Conference on Circuits and Systems.*, pages 145 – 148, December 2004.

[32] Jarkko Kari. Synchronizing finite automata on eulerian digraphs. *Theor. Comput. Sci.*, 295(1-3):223–232, 2003.

[33] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[34] Navin Kashyap and David L. Neuhoff. Data synchronization with timing: The variable-rate case. Submitted to IEEE Trans. Inform. Theory, June 2007.

[35] Shmuel T. Klein. Skeleton trees for the efficient decoding of Huffman encoded texts. *Inf. Retr.*, 3(1):7–23, 2000.

[36] Shmuel T. Klein and Miri Kopel Ben-Nissan. Using Fibonacci compression codes as alternatives to dense codes. In *Proc. 18th IEEE Data Compression Conference (DCC'08)*, pages 472–481, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[37] Shmuel T. Klein and Dana Shapira. Pattern matching in Huffman encoded texts. *Inf. Process. Manage.*, 41(4):829–841, 2005.

[38] Shmuel Tomi Klein and Yair Wiseman. Parallel Huffman decoding with applications to jpeg files. *Comput. J.*, 46(5):487–497, 2003.

[39] W.-M. Lam and A.R. Reibman. Self-synchronizing variable-length codes for image transmission. *IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP-92*, 3:477–480, 1992.

[40] Wai-Man Lam and S. R. Kulkarni. Extended synchronizing codewords for binary prefix codes. *IEEE Trans. Inform. Theory*, 42(3):984–987, May 1996.

[41] Dongyang Long, Weijia Jia, and Ming Li. Optimal synchronous coding. *Int. J. Comput. Math.*, 81(8):931–941, 2004.

[42] David J. C. Mackay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, June 2002.

[43] S. Malinowski, H. Jegou, and C. Guillemot. Error recovery properties and soft decoding of quasi-arithmetic codes. In *Proceedings of IEEE International Symposium on Information Theory 2006*, pages 2338–2342, July 2006.

[44] S. Malinowski, H. Jegou, and C. Guillemot. Synchronization recovery and state model reduction for soft decoding of variable length codes. *IEEE Trans. Inform. Theory*, 53(1):368–377, Jan. 2007.

[45] A. Mateescu and A. Salomaa. Many-valued truth functions, Černý's conjecture and road coloring. *Bulletin of the European Association for Theoretical Computer Science*, 68:134–, 1999.

[46] J. C. Maxted and John P. Robinson. Error recovery for variable length codes. *IEEE Trans. Inform. Theory*, 31(6):794–801, November 1985.

[47] Michael E. Monaco and James M. Lawler. Corrections and additions to 'error recovery for variable length codes' by J.C. Maxted and J.P. Robinson. *IEEE Trans. Inform. Theory*, 33(3):454–456, May 1987.

[48] Bruce L. Montgomery and Julia Abrahams. Synchronization of binary source codes. *IEEE Trans. Inform. Theory*, 32(6):849–854, November 1986.

[49] Hiroyoshi Morita, Adriaan J. de Lind van Wijngaarden, and A. J. Han Vinck. On the construction of maximal prefix-synchronized codes. *IEEE Trans. Inform. Theory*, 42(6):2158–2166, November 1996.

[50] Peter G. Neuman. Efficient error-limiting variable-length codes. *IRE Trans. Inform. Theory*, 8:292–304, July 1962.

[51] S. Perkins and A. E. Escott. Synchronizing codewords of q-ary Huffman codes. *Discrete Math.*, 197-198:637–655, 1999.

[52] S. Perkins and D. H. Smith. A scheme for the synchronization of variable length codes. *Discrete Appl. Math.*, 101(1-3):231–245, 2000.

[53] S. Perkins and D. H. Smith. Robust data compression: Variable length codes and burst errors. *The Computer Journal*, 48(3):315–322, 2005.

[54] S. Perkins, D. H. Smith, and A. Ryley. Robust data compression: Consistency checking in the synchronization of variable length codes. *The Computer Journal*, 47(3):309–319, 2004.

[55] Stephanie Perkins and Adrian Escott. Extended synchronizing codewords for q-ary complete prefix codes. *Discrete Mathematics*, 231(1-3):391–401, 2001.

[56] Jean-Eric Pin. On two combinatorial problems arising from automata theory. *Annals of Discrete Mathematics*, 17:535–548, 1983.

[57] M. Rahman and S. Misbahuddin. Effect of a Binary Symmetric Channel on the Synchronisation Recovery of Variable Length Codes. *The Computer Journal*, 32(3):246–251, 1989.

[58] David W. Redmill and Nick G. Kingsbury. The EREC: an error-resilient technique for coding variable-length blocks of data. *IEEE Transactions on Image Processing*, 5(4):565–574, 1996.

[59] Adam Roman. *Problemy synchronizacji automatów skończonych*. PhD dissertation, Jagiellonian University, 2006. In Polish.

[60] Beulah Rudner. Construction of minimum-redundancy codes with an optimum synchronizing property. *IEEE Trans. Inform. Theory*, 17(4):478–487, July 1971.

[61] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2007. With contributions by Giovanni Motta and David Bryant.

[62] David Salomon. *Variable-length Codes for Data Compression*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2007.

[63] Sven Sandberg. Homing and synchronizing sequences. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 5–33. Springer, 2004.

[64] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 3rd edition, December 2005.

[65] Marcel Paul Schützenberger. On synchronizing prefix codes. *Information and Control*, 11(4):396–401, 1967.

[66] Ahmed Al Soualhi and Essam Hassan. Simplified expression for the expected error span recovery for variable length codes. *Int. J. Electronics*, 75(5):811–816, 1993.

[67] Peter F. Swaszek and P. DiCicco. More on the error recovery for variable-length codes. *IEEE Trans. Inform. Theory*, 41(6):2064–2071, November 1995.

[68] Yasuhiro Takishima, Masahiro Wada, and Hitomi Murakami. Error states and synchronization recovery for variable length codes. *IEEE Trans. Communications*, 42(2/3/4):783–792, Feb/Mar/Apr 1995.

[69] Mark R. Titchener. The synchronization of variable-length codes. *IEEE Trans. Inform. Theory*, 43(2):683–691, 1997.

[70] A. N. Trahtman. An efficient algorithm finds noticeable trends and examples concerning the Černý conjecture. In Rastislav Kralovic and Pawel Urzyczyn, editors, *MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 789–800. Springer, 2006.

[71] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[72] Jiangtao Wen and J.D. Villasenor. Reversible variable length codes for robust image and video transmission. *1997. Conference Record of the Thirty-First Asilomar Conference on Signals, Systems and Computers*, 2(2-5):973–979, November 1997.

[73] Te-Chung Yang and Sunil Kumar. A low complexity error recovery technique for wavelet image codecs with inter-subband dependency. *IEEE Transactions on Consumer Electronics*, 48(4):973 – 981, November 2002.

[74] Guangcai Zhou and Zhen Zhang. Synchronization recovery of variable-length codes. *IEEE Trans. Inform. Theory*, 48(1):219–227, January 2002.