A STUDY OF SOLID HYPERCODES
(Thesis format: Monograph)

by

Taylor J. Smith

Undergraduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science

Faculty of Science
University of Western Ontario
London, Ontario, Canada

Abstract

Channel coding is a subarea of coding theory that focuses on methods of detecting and correcting errors in data. Depending on the methods being employed, the resulting codes that are used to transmit and receive data may vary greatly. Certain classes of codes possess certain unique properties, and these properties may be combined to create new classes of codes with stronger error resistance capabilities.

This thesis focuses on a class of codes known as the solid hypercodes. This class arises as the intersection of the sets of two other classes of codes: solid codes, where errors in any given word of a message do not compromise the correctness of the entire message, and hypercodes, where accidental insertions or deletions of symbols may be detected within individual words.

This thesis investigates the properties of solid hypercodes and, having established these properties, identifies the connections between solid hypercodes and similar codes. The work of a former graduate student is analyzed and improved upon in order to provide further insight into the structure and generation of these codes.

Keywords: coding theory, error correction, error detection, hypercodes, solid codes, solid hypercodes

Acknowledgements

To my supervisor, Dr. Helmut Jürgensen, thank you for your excellent guidance throughout the course of this research. Before I started this work, I knew nothing about coding theory. With your dedication and patience, however, you have enabled me to learn a great deal about this interesting topic, and I look forward to continuing my work with coding theory in the future. I would also like to thank you for your constant stream of ideas, which gave me much to think about, and for the time you spent translating the work that formed a large part of my thesis.

To the course administrator and undergraduate chair, Dr. Michael Katchabaw, thank you for giving me so many opportunities—not only within this thesis course, but also over the past four years I have spent at Western. With your direction and advice, I have been able to take part in many great things at this university.

To the faculty and staff of the Department of Computer Science, thank you for being constant sources of inspiration. The knowledge and assistance you have given me is immeasurable.

Finally, to my family and friends, thank you for your love, for your support, and for putting up with me every time I talked about my research.

Contents

List of Symbols

Σ	Finite alphabet
Σ^*	Set of all words over Σ
Σ^+	Set of all words over Σ, excluding the empty word
L	Language
$\|L\|$	Cardinality or size of a language
w	Word
$\|w\|$	Length of a word
$\|w\|_a$	Count of the symbol a within a word
ϵ	Empty word
$\emptyset$	Empty set

Chapter 1

Introduction

1.1 Overview of Coding Theory

Coding theory is the area of computer science that studies codes—a system of symbols and related rules—and their properties. Coding theory first emerged as an application of information theory, or the study of data communication, with Claude Shannon's 1948 paper on the mathematical basis for communication [13]. Over the years, research in coding theory has trended away from application and has become more aligned with research in mathematics and formal language theory, and the majority of current work continues to follow this trend. Today, there exist two primary subareas of coding theory: source coding, which discusses methods of compressing data, and channel coding, which aims to develop methods of detecting and correcting errors in data. This thesis concerns itself with the latter subarea.

The transmission of information may be represented by the following components: S, a source that sends information; γ, an information encoder; C, a physical channel such as a wire or a wave; δ, an information decoder; and R, a recipient. During transmission, the information may be affected by N, a source of noise. This system is depicted graphically in Figure 1.1. In general, the system is expected to work correctly, that is, for any message w, $\delta(\gamma(w)) = w$. However, the presence of noise may affect the system's performance and introduce errors [7].

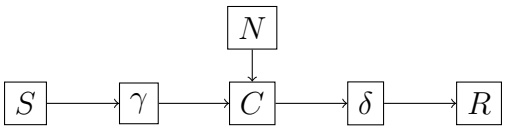

Figure 1.1: The information processing and transmission model

The existence of codes that are capable of detecting and correcting errors is vital to nearly all of the world's modern communication systems. Millions of messages are transmitted every day—from a cell phone receiving a call to the International Space Station sending a progress report—and these messages have one thing in common: they are all susceptible to noise and to corruption. The development of communication methods that ensure less noise, and thus a more reliable transmission, lies at the foundation of channel coding and forms the basis for this thesis.

1.2 Contributions of the Thesis

This thesis presents the following results.

Solid hypercodes are defined and a selection of properties relating to this class of codes are presented. The basis for an efficient method of checking the solid and hyper properties of a set of codes is established, and a number of tests are proposed to cover the majority of possible cases.

A method for generating sets of solid hypercodes is introduced. A detailed analysis of the method is presented, including a translation of the original source, a guide to understanding the code generation process, and a critique of some aspects of this process. A number of improvements to the original source code are discussed, and data and results stemming from these improvements are included. A complete translation of the original source code is included to facilitate future work on this topic.

Finally, some open questions and directions for future work are offered.

1.3 Chapter Outlines

This thesis is divided into five chapters, which discuss the following topics:

1 Introduction

The present chapter aims to provide the reader with a brief history of coding theory and an overview of the material contained within this thesis.

2 Background and Related Work

This chapter contains the mathematical preliminaries required to understand much of the material presented in this thesis. Additional sections which present different classes of error-detecting and error-correcting codes are included.

3 Solid Hypercodes

This chapter presents the main topic of this thesis. The concept of a solid hypercode is defined, and properties relating to this particular class of codes are developed and discussed.

4 Review of a Method to Enumerate Codes

This chapter introduces a code-generating software tool developed by a former student. The software tool is discussed from a methodological standpoint, and issues discovered within the source code of the tool are presented. Finally, improvements made to the source code are discussed along with results generated by the improved version of the software.

5 Conclusions

This chapter summarizes the results of the thesis and offers suggestions for future work relating to this topic.

Chapter 2

Background and Related Work

2.1 Preliminaries

A language L is a code if every string composed of words from L has a single decomposition. In other terms, every string may be represented in at most one way. A word from L has an L-factorization if, for some $n \geq 0$, the word can be written in discrete components as $u = u_1 u_2 \dots u_n$. With this information, a code may be defined as follows.

Definition 2.1 (Code)**.** A language $L \subseteq \Sigma^+$ is a code if every word $w \in L^+$ has a unique L-factorization, that is, $u_1 u_2 \dots u_m = v_1 v_2 \dots v_n$ with $u_i, v_j \in L$ for all i and j implies $m = n$ and $u_i = v_i$ for all i [17].

Different types of codes may offer different structures or sets of properties. One of the simplest properties stems from setting each string in the language to be a consistent length, which gives a block code.

Definition 2.2 (Block code)**.** A language L, defined as above, is a block code if every word in L has the same length [17].

Block codes are also known as "uniform codes" in some literature.

A number of other properties of codes can ensure that messages are decoded reliably. Two such properties are prefix-freeness and suffix-freeness, ensured by prefix codes and suffix codes respectively.

Definition 2.3 (Prefix code)**.** A language L, defined as above, is a prefix code if no word in L is a prefix of another word in L, that is, $L \cap L\Sigma^+ = \emptyset$ [17].

Definition 2.4 (Suffix code)**.** A language L, defined as above, is a suffix code if no word in L is a suffix of another word in L, that is, $L \cap \Sigma^+ L = \emptyset$ [17].

A word u is a subword of another word v if and only if $u = u_1u_2 \dots u_n$ and $v = v_1u_1v_2u_2 \dots v_nu_nv_{n+1}$ for some $n \geq 1$. The same word u is a proper subword of the same word v if $v_1 \dots v_{n+1} \neq \epsilon$.

2.2 Solid Codes

This section presents the first few classes of codes with non-trivial structures. The focus of this section is on the class of codes known as solid codes. Before defining solid codes, however, the concepts of infix codes and overlap-free codes must be discussed.

Infix codes consist of words that are independent of all other words in the code, that is, no word is contained within another word.

Definition 2.5 (Infix code)**.** A language $L \subseteq \Sigma^+$ is an infix code if words $v, uvw \in \Sigma^+$ are such that $v \in L$ and $uvw \in L$ implies that $u = \epsilon$ and $w = \epsilon$ [3].

Every infix code is both a prefix and a suffix code.

Overlap-free codes are exactly as the name suggests: codes containing words that, when placed together, do not have overlapping symbols. For instance, a word that ends with the symbol a may never be added to an overlap-free code containing a word that begins with the same symbol a, since the two words would then overlap.

Definition 2.6 (Overlap-free code)**.** A language $L \subseteq \Sigma^+$ is overlap-free if words $u, v, w \in \Sigma^+$ are such that $uv \in L$ and $vw \in L$ implies that one of $u, v,$ or w is equal to ϵ [3].

With the above definitions, the concept of a solid code may now be developed. Solid codes are error-resistant in the sense that errors within a given word of a received message will not affect the decoding of other correctly-received words. As a result, solid codes are very useful when transmitting information over a noisy channel. Levenshtein first studied solid codes in the early 1960s using the name “strongly regular codes", and later “codes without overlaps" [9, 10]. Shyr and Yu [16] introduced the concept of a solid code in the context of the study of disjunctive domains, and Jürgensen and Yu [8] proved that this approach was equivalent to Levenshtein's.

For a code to be considered solid, it must only meet the two conditions outlined below.

Definition 2.7 (Solid code)**.** A language $L \subseteq \Sigma^+$ is a solid code if it satisfies the following conditions [3]:

1. no word in L is a subword of a word in L (infix-freeness), and
2. no proper prefix of a word in L is a proper suffix of a word in L (overlap-freeness).

2.3 Hypercodes

The next important class of codes to be discussed are known as hypercodes. These codes, which were first studied by Shyr and Thierrin [14], are a special case of infix codes. Hypercodes are capable of detecting bursts of either insertions or deletions of symbols within a code word.

Definition 2.8 (Hypercode)**.** A language $L \subseteq \Sigma^+$ is a hypercode if no word in L is a proper subword of another word in L [14].

Often, the concept of the embedding order is used in discussions involving hypercodes. This embedding order, written as $\leq$, is identical to the definition of the subword, that is, $u \leq v$ if and only if $u = u_1u_2 \dots u_n$ and $v = v_1u_1v_2u_2 \dots v_nu_nv_{n+1}$ for some $n \geq 1$. Using this embedding order, a language $L \subseteq \Sigma^+$ is a hypercode if, for words $u, v \in L$, $u \leq v$ implies $u = v$ [11, 14].

The following theorem comes as a consequence of the definition of a hypercode.

Theorem 2.1. *Every block code is a hypercode.*

Proof. Proposition 5 of [14]. □

From Theorem 2.1, we know that solid block codes (and thus, solid hypercodes of fixed length) only need to meet the overlap condition. This relationship will be expanded upon in the following section.

2.4 Intercodes and Comma-Free Codes

This section takes a brief digression to introduce two related classes of codes that will help with the discussion of a result presented later in this chapter.

The concept of synchronization is essential to ensure the good performance of any communication system. When a system is synchronized, all parts of the system agree on the present state of the system [6]. Some systems incorporate a synchronization delay, which defines how many encoded symbols must be observed after an interruption before synchronization may be achieved. Different classes of codes have been developed to facilitate synchronization within a system.

An intercode of index n is a code that is synchronously decipherable with a delay less than or equal to n. Intercodes were first studied by Shyr and Yu [15]. Every intercode is a bifix code, where the bifix property ensures that any word belonging to a language is neither a prefix nor a suffix of any other word within that language.

Definition 2.9 (Intercode of index n)**.** A language $L \subseteq \Sigma^+$ is an intercode if, for some $n \geq 1$, $L^{n+1} \cap \Sigma^+ L^n \Sigma^+ = \emptyset$. In this case, the smallest such n is said to be the index of L [15].

A comma-free code is a particular type of intercode where the synchronous deciphering delay is equal to 1. The term "comma-free" means that no special symbol (often called a "comma") is necessary to ensure synchronization without delay. The notion of a comma-free code was originally introduced by Crick, Griffith, and Orgel [1] in a biological context, and was later defined as a mathematical object one year later by Golomb, Gordon, and Welch [2].

Definition 2.10 (Comma-free code)**.** The class of comma-free codes is exactly the class of intercodes of index 1 [15].

Hsieh, Hsu, and Shyr [5] showed that all comma-free codes are infix codes and that all solid codes are comma-free codes. However, it is not necessarily the case that all comma-free codes are solid. At the start of this research, the following case was investigated.

Proposition 2.2. *Every comma-free block code is a solid code.*

The following counterexample shows that this is not true.

Take the language $L = \{010\}$. Clearly, this language is a block code, and hence it is infix-free. However, the language is not overlap-free, so it is not a solid code. To check if L is comma-free, use the original definition of comma-freeness [2, 12]. By this definition, proper suffixes of code words are attached to proper prefixes of code words, and if the resulting words are not in L, then the language is comma-free.

For the single word in L, there are two proper suffixes (0 and 10) and two proper prefixes (0 and 01). To preserve the block code property of L, only words of length 3 are considered. These words are:

- suffix 0 followed by prefix 01, resulting in $001 \notin L$
- suffix 10 followed by prefix 0, resulting in $100 \notin L$

All other cases lead to words that are not of length 3, which are not in L. Thus, L is a comma-free block code which is not a solid code.

Chapter 3

Solid Hypercodes

3.1 Definition

Code words that are transmitted over a noisy channel are subject to many kinds of errors. For instance, symbols within a transmitted word may be altered, mistakenly inserted, or lost altogether. Fortunately, the previous section established that special classes of codes exist to counteract these errors. However, what would happen if an error occurred that a particular class of code was not equipped to handle? The error might remain unnoticed, thus resulting in a received message that is unknowingly flawed.

The solid hypercode is a class of codes that is designed to handle two types of errors at once. By using these codes, transmitted words become resistant to errors both between code words and within code words. As evidenced by the name, the definition of such a code is straightforward.

Definition 3.1 (Solid hypercode)**.** A language $L \subseteq \Sigma^+$ is a solid hypercode if it is both a solid code and a hypercode.

Much like the class of comma-free codes, solid hypercodes allow for synchronization with the added benefit of code words being handled independently. These properties stem from the solid aspect of the code. The hyper aspect, on the other hand, does not provide any synchronization benefits. Rather, it allows for a more thorough protection against errors.

3.2 Properties

Definition 3.1 shows that two properties of a code must be checked before it may be classified as a solid hypercode. The hyper property will be investigated first, followed by the solid property.

The most straightforward way of determining whether or not a language forms a hypercode is to determine whether one word within the language can be embedded into another. Often, the simplest method for determining this condition is to check if the given language is a block code. Theorem 2.1 establishes that every block code is also a hypercode.

In the case where words in a language are of unequal length (that is, the language is not a block code), other conditions must be checked. The easiest such check to make involves the embedding order of the words. Assume that, for words u and v, u is shorter than v.

Begin by storing words u and v as a sequence of letters in an array. Each position in each word is associated with a corresponding value to keep track of the number of occurrences of the given symbol from that position onwards. Each time a new word is generated and added to the set, these values are updated.

The property of overlap-freeness using a binary alphabet implies that words u and v start with the same symbol. If this simple test fails, then the entire check may be stopped. Otherwise, match these words and continue.

The next test involves the counts of each symbol within each word. The proposition below establishes one case that may be tested.

Proposition 3.1. *For all words $u, v \in \Sigma^+$ in a language L, where $|u| < |v|$, if there exists a symbol $a \in \Sigma$ such that $|u|_a > |v|_a$, then u cannot be embedded into v and this language forms a hypercode.*

If u can be embedded into v, then the test fails and the check may be stopped. Otherwise, move to the next symbol in u and find the next matching symbol in v. The fact that the test passes implies that these symbols exist, and so the process can continue. Match these letters and repeat.

Next, recall that for a language to form a solid code, it must be both infix-free and overlap-free. As a consequence of the definition of a hypercode, the infix-free property is satisfied, and so the only other property that must be checked is overlap-freeness. This check is trivial. If the first symbol of each word is different from the last symbol, then the language is overlap-free and is therefore a solid hypercode. If this condition does not hold, then the language is not a solid code.

There is still some work to be done in order to establish a robust method for checking the solid hypercode property in a reasonable amount of time. While the above remarks are a good start, the performance of this checking algorithm may be improved by using an appropriate data structure to store and check code words. In addition, the method above is only valid for binary alphabets, though it may be possible to adapt this work to the non-binary case.

Chapter 4

Review of a Method to Enumerate Codes

4.1 Overview of Method

A major portion of the work completed in this thesis was based on the related work of a former graduate student. Christian Herrmann's thesis [4] was written in 2005 at Potsdam University under the supervision of Dr. Helmut Jürgensen. The title of the thesis, translated to English, is "Development of Methods for the Enumeration and Investigation of Solid Codes, Hypercodes, and Solid Hypercodes".

In Herrmann's thesis, he developed a software tool to generate examples of sets of solid hypercodes. Although the software tool was functional, the algorithm used to find such sets was very slow. Thus, part of this research was dedicated to understanding and improving this software tool in the hope of achieving a more efficient program that is capable of generating larger sets of solid hypercodes.

The thesis begins by defining relevant classes of codes, including the solid hypercodes. A chapter on the enumeration of codes follows, where different enumeration algorithms are presented and discussed. Dedicated chapters on solid codes, hypercodes, and solid hypercodes constitute the rest of the thesis, and the features, properties, and generation methods of each of these classes of codes are included in their respective chapters. Brief discussions on the detection and correction of errors, as well as on synchronization properties, are interspersed throughout the thesis.

4.2 Analysis of Method

The initial method used to generate sets of solid hypercodes is detailed in Chapter 3 of Herrmann's thesis. This method will be presented and discussed before the changes to the method are introduced. Where applicable, the original German names for classes and variables are included beside their English translations.

In Section 3.1, "Enumeration of Code Words", Herrmann outlines a basic technique for obtaining code words and generating new code words given previous code words. The `CCodeWord::Next()` (`CCodeWort::Vor()`) method is introduced in Listing 1 and discussed throughout the section. The `letter` (`buchstabe`) variable contains the given code word indexed by symbol. The `length` (`laenge`) variable represents the given code word's length, and `position` points to the current symbol being analyzed within the code word. The leading symbol in the code word is located at position 1. In the event of an overflow within a code word, a new word that is one bit longer will be generated.

Section 3.2, "Enumeration of Codes", presents a technique to ensure that every possible code word is generated for a given input value. To avoid duplicate code words, the generation process constructs larger words first in descending lexicographical order. This ordering is evident in Tabelle 3, where the possible code words in the top row are sorted in descending order, that is, `1` is to the left of `0`, `00` is to the left of `1`, and so on. The rows underneath these possible code words are binary values indicating whether or not a particular code word is included in the given set.

The first algorithm to enumerate codes is developed in Section 3.3, "Enumeration of Codes—Algorithm I". With this algorithm, a pointer variable `word` (`wort`) is allocated to contain an array of instances of `CCodeWord`s. The algorithm itself, found in Listing 2, performs the actual enumeration, where `x` represents the current number of code words in the set. The idea behind this algorithm is to generate all code words, then test each code word for the properties of a solid hypercode. The algorithm looks at two cases:

- If the last word in the code is the smallest code word, but this is not the only word in the code, then add another word by scanning the code for an "empty" spot and placing the word in that spot. If not, then take the next word to be the code.

- Otherwise, add a new word with position 1.

Unfortunately, this first iteration of the algorithm suffers from severe performance issues. A number of redundant computations are found throughout the implementation. Most notably, the following quality is overlooked:

Remark. Given a property P of finite languages L and L' over an alphabet Σ, where $L \subseteq L' \subseteq \Sigma^*$, if $\neg P(L) \Rightarrow \neg P(L')$, then no superset of L needs to be tested for P.

As a result of this oversight, the algorithm is slowed down by redundancy.

To improve the performance of the enumeration algorithm, a revised version is developed in Section 3.4, "Enumeration of Codes—Algorithm II". This iteration of the algorithm makes use of backtracking, or enumerating all possibilities in a tree and taking one step backwards within the tree if an incorrect result is found. Using this method, one can assume that if n code words are found that meet the required properties, then each nonempty subset is a set of codes, and a total of $2^n - 1$ sets of codes are thus produced. This iteration of the algorithm appears to use breadth-first searching to enumerate codes, as illustrated in Abbildung 1.

The `CCode::RecursionStart()` (`CCode::RekursionStarten()`) method in Listing 3 initializes the enumeration process, while the `CCode::Recursion()` (`CCode::Rekursion()`) method in Listing 4 continues the recursive enumeration with a `depth` (`tiefe`) of 2. Again, the code words are stored in the pointer variable `word`. The algorithm begins by checking if the current word starts with the symbol `1` and ends with the symbol `0`. If this condition holds, then the code word is checked against itself for the existence of overlaps. If no overlaps are found, then the word forms the basis for the tree and the recursion continues. The running time of this iteration of the algorithm is proportional to `depth`.

Section 3.5 concludes with some remarks on alphabets and the equality of codes.

4.3 Improvements

The original source code underwent some changes to improve performance and to allow for future research into this topic. This section lists a summary of the changes. The most recent version of the source code may be found in Appendix A of this thesis.

The source code of the software tool, including comments and variable names, was initially written in German. Each line of the source code has been translated to English to allow for easier readability and increased portability. A number of comments have also been added throughout the source code. Along with these cosmetic improvements, the formatting and spacing within each file has been adjusted to bring stylistic consistency to the source code.

In terms of performance, minor changes were made to the methods responsible for the generation of code words. In particular, the methods `CCodeWord::Next()` and `CCodeWord::Previous()` were altered to reduce redundant computation. These changes

resulted in a slight decrease in the run time of the program, though the change was not substantial.

Table 4.1 lists the run times of both the original and the modified main program on different input values[1]. Note that no input value below 400 was included in the table, since the run times for such small input values were negligible. Due to rounding errors, time values close to 1 second may appear as "0".

Initially, the main program responsible for generating sets of codes (`Diplomarbeit.cpp`) required the user to change a hardcoded value within the source code in order to obtain different results. This program was altered to allow the user to input a boundary value at runtime instead, thus allowing results to be obtained more easily. In addition to this modified input method, the program is now capable of outputting detailed information to the screen, including the start and end times of the program as well as the boundary value and last code generated for that value.

A secondary program (`CodeReader.cpp`) was developed to allow users to generate examples of sets of solid hypercodes where the difference in length between code words in any given set is greater than or equal to 2. By keeping only these particular sets, certain properties of the class of solid hypercodes can be investigated to determine whether or not code word length plays a role in how these codes behave. Indeed, much of the work presented in Chapter 3 of this thesis came from the analysis of sets of codes generated by this program. Figure 4.1 presents a sampling of such codes.

A number of changes have yet to be completed in order to further improve the performance of the generation algorithm. For instance, a check could be implemented to compare generated code words to existing code words, which would prevent identical code words from being generated and reduce the overall redundancy of the program. In the long term, it may be possible to remove a factor of `length` from the run time of the program by modifying the `letter[i]` variable to return the i-th bit of the code word, thus only requiring the `CCodeWord::Next()` method to increment `position`. However, the consequences of making such a change have not yet been investigated.

[1]This data was collected by running `Diplomarbeit.cpp` on the department server `algernon.csd.uwo.ca`. The server has two single-core 900MHz UltraSPARC-III+ CPUs and 3GB of RAM.

Input	Original	Modified
400	1	0
450	0	1
500	28	26
550	39	38
600	39	38
650	39	37
700	39	38
750	39	38
800	39	38
850	52	50
900	166	158
950	1052	1013

Table 4.1: Comparative run times of `Diplomarbeit.cpp` (in seconds)

```
1010000 11100       1111100  11010
1100000 10100       10100000 11100
1100000 11010       10100000 101100
1110000 10100       10100000 111000
1110000 11010       10100000 111100
1111000 10100       10100100 111000
1111000 11010       10100100 111100
1111010 11000       10101000 111100
1111100 10100       10101100 111000
```

Figure 4.1: A sample of two-word solid hypercodes generated by `CodeReader.cpp`

Chapter 5

Conclusions

This thesis defined the class of solid hypercodes, presented a selection of properties relating to these codes, and established the basis for an efficient method of checking both the solid and the hyper properties of a set of codes. A method of generating sets of solid hypercodes was also discussed alongside a number of changes that were made to the method. These changes resulted in a slight performance increase, but some additional possibilities for improvement were identified.

The class of solid hypercodes continues to grow as a topic of study, and although this thesis offers many new contributions, there remains a lot of work to be done. At the end of Chapters 3 and 4, a number of open questions are established which could direct future research on this topic, be it theoretical (relating to the mathematical concept of solid hypercodes) or applied (relating to the solid hypercode generation method).

Appendix A

Source Code

CCode.cpp

```
1  /**
2   * CCode.cpp
3   * Contains methods for operations between codewords
4   * Written by Christian Herrmann, 2005
5   * Modified by Taylor J. Smith, 2015
6   */
7
8  #include <iostream>
9  #include "CCode.h"
10
11 // Constructor
12 CCode::CCode() {
13     CCodeWord * cw = new CCodeWord();
14     this->word = new CCodeWord[(1+1)];
15     this->word[1] = *cw;
16     this->number = 1;
17     this->position = 1;
18 }
19
20 // Destructor
21 CCode::~CCode() {
22     this->word->~CCodeWord();
23     delete[] this->word;
24 }
25
26 // Next code
27 void CCode::Next() {
28     int x = this->number;
29     bool done = false;
30
31     if (this->word[x].getPosition() == 1) {
32         while(!done) {
33             if (x > 1) {
34                 if ((this->word[(x-1)].getPosition() - this->word[x].getPosition()) > 1) {
35                     this->word[x].Next();
36                     this->number = x;
37                     done = true;
38                 } else {
39                     if (this->word[x-1].getPosition() - this->word[x].getPosition() == 1) {
40                         delete[] this->word[x].letter;
41                         this->word[x].letter = new bool[(1+1)];
42                         this->word[x].letter[1] = 0;
43                         this->word[x].setPosition(1);
44                         this->word[x].setLength(1);
45                         x--;
46                     }
47                 }
48             } else {
```

```
49                  if (x == 1) {
50                      this->word[x].Next();
51                      this->number = x;
52                      done = true;
53                  }
54              }
55          }
56      } else {
57          if (this->word[x].getPosition() > 1) {
58              CCodeWord* temp = new CCodeWord[((x+1)+1)];
59
60              for (int k = 1; k <= (x); k++) {
61                temp[k] = this->word[k];
62              }
63
64              delete[] this->word;
65              this->word = new CCodeWord[((x+1)+1)];
66
67              for (int k = 1; k <= (x); k++) {
68                this->word[k] = temp[k];
69              }
70
71              delete[] temp;
72              delete[] this->word[x+1].letter;
73              this->word[x+1].letter = new bool[(1+1)];
74              this->word[x+1].letter[1] = 0;
75              this->word[x+1].setPosition(1);
76              this->word[x+1].setLength(1);
77              x++;
78              this->number = x;
79          }
80      }
81      this->number = x;
82  }
83
84  // Previous code
85  void CCode::Previous() {
86  }
87
88  // Set number
89  void CCode::setNumber(int num) {
90      this->number = num;
91  }
92
93  // Set position
94  void CCode::setPosition(int pos) {
95      this->position = pos;
96  }
97
98  // Get number
99  int CCode::getNumber() {
100     return this->number;
101 }
102
103 // Test whether the code is solid
104 bool CCode::isSolidCode() {
105     int start = 0;
106     int end = 0;
107     int neg = -1;
108
109     for (int i = 1; i <= this->number; i++) {
110       for (int l = i; l <= this->number; l++) {
111         if (i != l) {
112           start = this->word[l].getLength();
113           start = start * neg;
114           start = start + 1;
115
116           end = this->word[i].getLength();
117           end = end - 1;
118
119           for (int t = start; t <= end; t++) {
120             if (!((this->word[i].getPosition() == this->word[l].getPosition()) && (t == 0))) {
121               if (this->Comparison(this->word[i], this->word[l], t)) {
122                 return false;
```

```
123                         }
124                     }
125                 }
126             }
127         }
128     }
129
130     return true;
131 }
132
133 // Test whether the code is solid
134 bool CCode::isSolidCodeWord(const CCodeWord& codew1, const CCodeWord& codew2) {
135     int start = (codew2.getLength() * (-1)) + 1;
136     int end = codew1.getLength() - 1;
137
138     for (int t = start; t <= end; t++) {
139       if (!((codew1.getPosition() == codew2.getPosition()) && (t == 0))) {
140         if (this->Comparison(codew1, codew2, t)) {
141           return false;
142         }
143       }
144     }
145
146     return true;
147 }
148
149 // Test whether the code is hyper
150 bool CCode::isHyperCode() {
151     bool isHyperC;
152
153     for (int i = 1; i <= this->number; i++) {
154       for (int l = i+1; l <= this->number; l++) {
155         isHyperC = this->isHyperCodeWord(this->word[l],this->word[i]);
156
157         if ( !isHyperC ) {
158           return false;
159         }
160       }
161     }
162
163     return true;
164 }
165
166 // Test whether the code is hyper
167 bool CCode::isHyperCodeWord(const CCodeWord& codew1, const CCodeWord& codew2) {
168     bool isHyperCW;
169
170     if (codew1.getLength() == 0) {
171       return false;
172     }
173
174     for (int i = 1; i <= (codew2.getLength() - codew1.getLength()+1); i++) {
175       if (codew1.letter[1] == codew2.letter[i]) {
176         isHyperCW = isHyperCodeWord(Remove(1, codew1), Remove(i, codew2));
177         if ( !isHyperCW ) {
178           return false;
179         }
180       }
181     }
182
183     return true;
184 }
185
186 // Start recursion
187 void CCode::RecursionStart(int end) {
188     for (int i = 1; i < end; i++) {
189       if (First1Last0(this->word[1])) {
190         if(isSolidCodeWord(this->word[1], this->word[1])) {
191           this->Display();
192           this->Recursion(2);
193         }
194       }
195
196       this->word[1].Next();
```

```
197     }
198 }
199
200 // Recursion to enumerate codes
201 void CCode::Recursion(int depth) {
202     bool a = true;
203     bool b = true;
204
205     CCodeWord* temp = new CCodeWord[(depth+1)];
206
207     for (int k = 1; k < depth; k++) {
208       temp[k] = this->word[k];
209     }
210
211     delete[] this->word;
212     this->word = new CCodeWord[(depth+1)];
213
214     for (int k = 1; k < depth; k++) {
215       this->word[k] = temp[k];
216     }
217
218     this->number = depth;
219
220     delete[] temp;
221     delete[] this->word[depth].letter;
222
223     this->word[depth].letter = new bool[(1+1)];
224     this->word[depth].letter[1] = 0;
225     this->word[depth].setPosition(1);
226     this->word[depth].setLength(1);
227
228     for (int i = 1; i < this->word[depth-1].getPosition(); i++) {
229       if (First1Last0(this->word[depth])) {
230         for (int l = 1; l <= depth; l++) {
231           a = isSolidCodeWord(this->word[l], this->word[depth]);
232
233           if (a && (depth != l)) {
234             a = isHyperCodeWord(this->word[depth], this->word[l]);
235           }
236
237           if (!a) {
238             l = depth+1;
239             b = false;
240           }
241         }
242
243         if (b) {
244           this->Display();
245           this->Recursion(depth+1);
246         }
247       }
248
249       this->word[depth].Next();
250       a = true;
251       b = true;
252     }
253
254     // After recursion, the number must be reduced by 1
255     this->number = depth-1;
256 }
257
258 // Display
259 void CCode::Display() {
260     for (int i = 1; i <= this->number; i++) {
261         this->word[i].Display();
262         std::cout << " ";
263     }
264
265     std::cout << "" << std::endl;
266 }
267
268 // Compares the overlapping parts of two codewords
269 bool CCode::Comparison(const CCodeWord& codew1, const CCodeWord& codew2, int shift) {
270     int start;
```

```
271     int end;
272
273     if (shift > 0) {
274       start = shift+1;
275     } else {
276       start = 1;
277     }
278
279     if (codew1.getLength() < (codew2.getLength()+shift)) {
280       end = codew1.getLength();
281     } else {
282       end = codew2.getLength()+shift;
283     }
284
285     for (int i=start; i<=end; i++) {
286       if (codew1.letter[i] != codew2.letter[(i-shift)]) {
287         return false;
288       }
289     }
290
291     return true;
292 }
293
294 // Returns the codeword without the first "count" symbols
295 CCodeWord CCode::Remove(int count, const CCodeWord& codew) {
296     CCodeWord codewBack = CCodeWord();
297
298     if ((codew.getLength() - count) <= 0) {
299       codewBack.setLength(0);
300       codewBack.setPosition(0);
301       return codewBack;
302     } else {
303       codewBack.setLength((codew.getLength() - count));
304       delete[] codewBack.letter;
305       codewBack.letter = new bool[codewBack.getLength()];
306
307       for(int i = count+1; i <= codew.getLength(); i++) {
308         codewBack.letter[i-count] = codew.letter[i];
309       }
310
311       return codewBack;
312     }
313 }
314
315 // Checks whether the codeword starts with 1 and ends with 0
316 bool CCode::First1Last0(const CCodeWord& codew) {
317     if (codew.getLength() == 1) {
318       return true;
319     }
320
321     if (codew.letter[1] == 1) {
322       if (codew.letter[codew.getLength()] == 0) {
323         return true;
324       }
325     }
326
327     return false;
328 }
```

CCode.h

```
/**
 * CCode.h
 * Header file for CCode.cpp
 * Written by Christian Herrmann, 2005
 * Modified by Taylor J. Smith, 2015
 */

#include "CCodeWord.h"

class CCode {
  public:
    // Constructor
    CCode();

    // Destructor
    ~CCode();

    // Next codeword
    void Next();

    // Previous codeword
    void Previous();

    // Set number
    void setNumber(int);

    // Set position
    void setPosition(int);

    // Get number
    int getNumber();

    // Test if this codeword is a solid codeword
    bool isSolidCode();

    // Test if the two codewords are solid codewords
    bool isSolidCodeWord(const CCodeWord&, const CCodeWord&);

    // Test if this codeword is a hyper codeword
    bool isHyperCode();

    // Test if the two codewords are hyper codewords
    bool isHyperCodeWord(const CCodeWord&, const CCodeWord&);

    // Start recursion
    void RecursionStart(int);

    // Recursion to enumerate codes
    void Recursion(int);

    // Display
    void Display();

    // Comparison of overlapping parts of two codewords
    bool Comparison(const CCodeWord&, const CCodeWord&, int);

    // Remove the first "int" symbols from the codeword
    CCodeWord Remove(int, const CCodeWord&);

    // Check whether codeword starts with 1 and ends with 0
    bool First1Last0(const CCodeWord&);

    CCodeWord* word;

    private:
    int number;
    int position;
};
```

CCodeWord.cpp

```
/**
 * CCodeWord.cpp
 * Contains methods for operations within codewords
 * Written by Christian Herrmann, 2005
 * Modified by Taylor J. Smith, 2015
 */

#include <iostream>
#include "CCodeWord.h"

// Constructor
CCodeWord::CCodeWord() {
    this->letter = new bool[1+1];
    this->letter[1] = 0;
    this->length = 1;
    this->position = 1;
}

// Destructor
CCodeWord::~CCodeWord() {
    delete[] this->letter;
}

CCodeWord& CCodeWord::operator=(const CCodeWord& cw) {
    this->length = cw.getLength();
    this->position = cw.getPosition();
    delete[] this->letter;
    this->letter = new bool[(this->length+1)];

    for(int i = 1; i <= this->length; i++) {
      this->letter[i] = cw.letter[i];
    }

    return *this;
}

// Get codeword length
int CCodeWord::getLength() const {
    return this->length;
}

// Get codeword position
int CCodeWord::getPosition() const {
    return this->position;
}

// Set codeword length
void CCodeWord::setLength(int l) {
    this->length = l;
}

// Set codeword position
void CCodeWord::setPosition(int p) {
    this->position = p;
}

// Next codeword
void CCodeWord::Next() {
    bool breakCheck = 0;

    for(int i = this->length; i >= 1; i--) {
        if (this->letter[i]) {
            this->letter[i] = 0;
        } else {
            this->letter[i] = 1;
            breakCheck = 1;
            break;
        }
    }

    if (breakCheck) {
```

```
72          this->position++;
73      } else {
74          this->length++;
75          delete[] this->letter;
76          this->letter = new bool[this->length+1];
77
78          for(int l = 1; l <= this->length; l++) {
79              this->letter[l] = 0;
80          }
81
82          this->position++;
83      }
84  }
85
86  // Previous codeword
87  void CCodeWord::Previous() {
88      bool breakCheck = 0;
89
90      if (this->position != 1) {
91          for(int i = this->length; i >= 1; i--) {
92              if (!this->letter[i]) {
93                this->letter[i] = 1;
94              } else {
95                this->letter[i] = 0;
96                breakCheck = 1;
97                break;
98              }
99          }
100
101         if (breakCheck) {
102             this->position--;
103         } else {
104             this->length--;
105             this->position--;
106         }
107     }
108 }
109
110 // Display
111 void CCodeWord::Display() const {
112     for(int i = 1; i <= this->length; i++) {
113         std::cout << this->letter[i];
114     }
115 }
116
117 // Display information
118 void CCodeWord::DisplayInfo() {
119     for(int i=1; i<=this->length; i++) {
120         std::cout << this->letter[i];
121     }
122
123     std::cout << "" << std::endl;
124     std::cout << this->position << std::endl;
125     std::cout << this->length << std::endl;
126     std::cout << "" << std::endl;
127 }
```

CCodeWord.h

```
/**
 * CCodeWord.h
 * Header file for CCodeWord.cpp
 * Written by Christian Herrmann, 2005
 * Modified by Taylor J. Smith, 2015
 */

class CCodeWord {
  public:
    // Constructor
    CCodeWord();

    // Destructor
    ~CCodeWord();

    CCodeWord& operator=(const CCodeWord&);

    // Output codeword length
    int getLength() const;

    // Output of codeword position
    int getPosition() const;

    // Set codeword length
    void setLength(int);

    // Set codeword position
    void setPosition(int);

    // Next codeword
    void Next();

    // Previous codeword
    void Previous();

    // Display
    void Display() const;

    // Display information
    void DisplayInfo();

    bool* letter;

  private:
    int length;
    int position;
};
```

Diplomarbeit.cpp

```
/**
 * Diplomarbeit.cpp
 * Main program to generate sets of solid hypercodes
 * Written by Christian Herrmann, 2005
 * Modified by Taylor J. Smith, 2015
 */

#include <time.h>
#include <iostream>
#include <fstream>
#include <string>
#include "CCode.h"

int main(int argc, char* argv[]) {
    time_t time1;           // time 1
    time_t time2;           // time 2
    double distance;        // distance

    // This variable forms the end condition
    int bound;

    if (argc != 2) {
        std::cout << "Improper number of arguments." << std::endl;
    } else {
        bound = atoi(argv[1]);
    }

    std::ofstream out("output.txt");                // create output file
    std::streambuf *coutbuf = std::cout.rdbuf();    // save old buf
    std::cout.rdbuf(out.rdbuf());                   // redirect standard output to file

    time(&time1);
    CCode test;

    test.RecursionStart(schl);

    time(&time2);
    distance = difftime(time2,time1);

    std::cout.rdbuf(coutbuf);                       // reset standard output

    // More detailed output
    std::cout << "File saved." << std::endl;
    std::cout << "Start time: " << ctime(&time1) << std::endl;
    std::cout << "End time: " << ctime(&time2) << std::endl;

    std::cout << "The start of the loop is at " << bound << "." << std::endl;
    std::cout << "Last code: ";
    test.Display();

    // Less detailed output
    // std::cout << "Input value: " << schl << ". Runtime: " << distance << " seconds." << std::endl;
}
```

CodeReader.cpp

```
/**
 * CodeReader.cpp
 * Secondary program to generate sets of solid hypercodes of unequal length
 * Written by Taylor J. Smith, 2015
 */

#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main(int argc, char* argv[]) {
    /* prepare input file */
    ifstream inputFile(argv[1]);

    /* check if file opened successfully */
    if (inputFile.is_open()) {
        string currLine;
        string currWord;
        int currLength;
        int firstLength;
        int count = 0;
        bool lengthTest = false;

        /* prepare output file */
        ofstream outputFile("modOutput.txt");

        /* while there are lines to be read */
        while (getline(inputFile, currLine)) {
            /* place line into stream */
            stringstream ss(currLine);

            /* while there are words to be read */
            while (getline(ss, currWord, ' ')) {
                /* get current word length */
                currLength = currWord.length();

                /* if this is the first word */
                if (count == 0) {
                    /* remember first word length */
                    firstLength = currWord.length();
                }

                /* get difference of word lengths */
                int diff = abs(firstLength-currLength);

                /* if difference is greater than specified threshold */
                if (diff > 1) {
                    /* remember this fact */
                    lengthTest = true;
                }

                ++count;
            }

            /* if lengths of words are not equal */
            if ((lengthTest) && (count > 1)) {
                /* place line into output file */
                outputFile << currLine << endl;
            } else {
                /* do nothing */
                continue;
            }

            /* reset for next iteration */
            count = 0;
            lengthTest = false;
        }

```

```
72          /* close files */
73          inputFile.close();
74          outputFile.close();
75      } else {
76          /* print error message */
77          cout << "File error" << endl;
78      }
79  }
```

Bibliography

[1] Francis H. C. Crick, John S. Griffith, and Leslie E. Orgel. Codes without commas. *Proceedings of the National Academy of Sciences*, 43(5):416–421, 1957.

[2] Solomon W. Golomb, Basil Gordon, and Lloyd R. Welch. Comma-free codes. *Canadian Journal of Mathematics*, 10:202–209, 1958.

[3] Yo-Sub Han and Kai Salomaa. Overlap-free languages and solid codes. *International Journal of Foundations of Computer Science*, 22(5):1197–1209, 2011.

[4] Christian Herrmann. Entwicklung von Methoden zur Aufzählung und Untersuchung von soliden Codes, Hyper-Codes und soliden Hyper-Codes. Diplomarbeit, Universität Potsdam, 2005. In German.

[5] C. Y. Hsieh, S. C. Hsu, and Huei-Jan Shyr. Some algebraic properties of comma-free codes. *RIMS Kenkyuroku*, 697:57–66, 1989.

[6] Helmut Jürgensen. Synchronization. *Information and Computation*, 206(9–10):1033–1044, 2008.

[7] Helmut Jürgensen and Stavros Konstantinidis. Codes. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 511–607. Springer-Verlag, Berlin Heidelberg, 1997.

[8] Helmut Jürgensen and Shyr-Shen Yu. Solid codes. *Journal of Information Processing and Cybernetics*, 26(10):563–574, 1990.

[9] Vladimir I. Levenshtein. Декодирующие автоматы, инвариантные относительно начального состояния. *Problemy Kibernetiki*, 12:125–136, 1964. In Russian.

[10] Vladimir I. Levenshtein. О максимальном числе слов в кодах без перекрытий. *Problemy Peredachi Informatsii*, 6(4):88–90, 1970. In Russian.

[11] Helmut Prodinger and Gabriel Thierrin. Towards a general concept of hypercodes. *Journal of Information and Optimization Sciences*, 4(3):255–268, 1983.

[12] Robert A. Scholtz. Maximal and variable word-length comma-free codes. *IEEE Transactions on Information Theory*, 15(2):300–306, 1969.

[13] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, 1948.

[14] Huei-Jan Shyr and Gabriel Thierrin. Hypercodes. *Information and Control*, 24(1):45–54, 1974.

[15] Huei-Jan Shyr and Shyr-Shen Yu. Intercodes and some related properties. *Soochow Journal of Mathematics*, 16(1):95–107, 1990.

[16] Huei-Jan Shyr and Shyr-Shen Yu. Solid codes and disjunctive domains. *Semigroup Forum*, 41:23–37, 1990.

[17] Shyr-Shen Yu. *Languages and Codes.* Tsang Hai Book Publishing Company, Taichung, 2005.