Finding Synchronization Codes to Boost Compression by Substring Enumeration

Dany Vohl, Claude-Guy Quimper, and Danny Dubé

Université Laval, Canada
Dany.Vohl.1@ulaval.ca
Claude-Guy.Quimper@ift.ulaval.ca
Danny.Dube@ift.ulaval.ca

Abstract. Synchronization codes are frequently used in numerical data transmission and storage. Compression by Substring Enumeration (CSE) is a new lossless compression scheme that has turned into a new and unusual application for synchronization codes. CSE is an inherently bit-oriented technique. However, since the usual benchmark files are all byte-oriented, CSE incurred a penalty due to a problem called *phase unawareness*. Subsequent work showed that inserting a synchronization code inside the data *before* compressing it improves the compression performance. In this paper, we present two constraint models that compute the shortest synchronization codes, i.e. those that add the fewest synchronization bits to the original data. We find synchronization codes for blocks of up to 64 bits.

1 Introduction

Synchronization codes are frequently used in numerical data transmission and storage [1–6]. While they might not be as well known as error-correction and error-detection codes, they still play a crucial role. Indeed, whenever the reception of data becomes ill-synchronized, data gets distorted. In such a case, even error-correction codes are of no help since they might interpret some data symbols as control symbols and vice-versa. In almost every application, keeping the transmission and storage of data correctly synchronized is considered to be a separate, lower-level task, which is handled by synchronization codes.

Recent work on data compression gives synchronization codes a new and rather unusual purpose [7, 8]. These are used to *boost* the performance of a specific data compressor. A preprocessing step is added to the data compressor which consists in inserting synchronization codes to the data. Since inserting a synchronization code causes the data to expand, it seems at first glance that this preprocessing is counter-productive. However, in this specific setup, the insertion of the synchronization codes improves the performance of the subsequent compression step, as measured (in an absolute sense) on the compressed data.[1]

[1] That is, the size of the synchronized-and-compressed data is smaller than the size of the directly compressed data, in an absolute sense. We insist on the term "absolute"

The data compression technique is called *Compression by Substring Enumeration* (CSE) [9].

In order to be successful as compression boosters, the considered synchronization codes must obey some properties. In particular, it is desirable for the codes not to cause too big an expansion of the data. Indeed, the subsequent compression phase has to compensate for all of this expansion. Also, the important characteristics of the synchronization codes is their synchronization power; i.e. how effectively they provide synchronization information. In the light of these goals, this paper aims at designing strong synchronization codes that expand data as little as possible.

The paper is structured as follows. Section 2 briefly surveys the synchronization codes and formally defines the family of codes that we focus on. Section 3 presents the key features of CSE and gives some intuitive reasons why CSE has the potential to be boosted by synchronization codes. Next, we present two constraint models that we use to compute a synchronization code. Section 4 presents a model that is based on constraint programming. Section 5 presents a pseudo-Boolean model. Section 6 discusses about ways to break symmetries among the sets of codes that are considered in order to reduce the time spent by the solvers in searching. The experiments described in Section 7 produced synchronization codes for words of 2 to 8, 16, 32, and 64 bits. Our codes are proved optimal for words of 2 to 8, 16 and 32 bits.

2 Synchronization Codes

2.1 Overview

There exists a wide variety of synchronization codes. Codes differ according to the properties they feature and the principles they are based on. Naturally, their effectiveness depends on the application at hand. We start by illustrating the need for synchronization codes by considering two examples of applications.

Our first application is a serial communication link. In serial communication, symbols (typically bits) are transmitted one after the other through a channel from an emitter to a receiver. When the pace of transmission is controlled by a clock, it might be the case that, in fact, the emitter and the receiver each have their own separate clock. In such a context, it is likely that the clocks do not have exactly the same frequencies. If the receiver's clock is slower than the emitter's clock, then bits might go undetected from time to time by the receiver. On the other hand, if the receiver's clock is faster, then a single bit sent by the emitter might get sampled twice by the receiver. Such communication errors are called synchronization errors. Naturally, some form of synchronization mechanism has to be provided in order to prevent these errors. In some contexts, the designer of the communication device might have the luxury to join a control channel to the data channel (e.g., on the motherboard of a computer). The control channel

here since making synchronized data more compressible, in a *relative* sense, is trivial because one only needs to stuff the data with highly repetitive paddings.

transmits synchronization symbols while the data channel transmits the payload symbols. In other contexts, no separate control channel is available and the synchronization information must be inserted within the payload data.

Our second application is a hard disk. In a hard disk, the read/write (RW) head hovers above one of the tracks of the spinning surface. The RW process of the hard disk is subject to the same synchronization errors as the serial communication link. For instance, serial-like synchronization errors are possible due to the imprecision in the rotation speed of the disk or some other factor. But the RW process also faces additional challenges. Indeed, the RW head has to determine where a track starts; in particular, after it moves from one track to another. Some sort of marker has to tell the RW head where the track starts (or, at a finer level, where the individual sectors of the track start). Nowadays, the magnetic bits are so densely packed on the tracks that it is unrealistic to rely on a physical device to mark the beginning of the tracks. Instead, markers identifying the start of the tracks (or sectors) have to be integrated among the magnetic information. Since waiting for a unique "track beginning" marker would cause a waste of time, we are likely to require the inscription of more frequent "sector beginning" markers. These enable quicker recovery of the synchronization and also offer the opportunity to join a header that includes additional useful information such as the sector IDs.

2.2 Characteristics of Synchronization Codes

Below is a list of considerations (either requirements or features) about synchronization codes. But before we give the list, we introduce a few definitions and a (not completely general) notation for synchronization codes that we use throughout the paper.

A sequence $A = a_0 a_1 \dots a_{|A|-1}$, is an ordered list of characters taken from an alphabet Σ. The length of a sequence A is the number of characters in the sequence and is denoted $|A|$. The subsequence $A[i..j]$ is formed by the characters $a_i a_{i+1} \dots a_{j-1} a_j$. The *rotation by one*[2] of a sequence is obtained by deleting the first character of the sequence and appending it to the end of the sequence. For instance, the rotation by one of the sequence `abcd` is `bcda`. We denote by A^1 the rotation by one of a sequence A. A *rotation by i* of a sequence A, which we denote by A^i, is obtained by applying i times a rotation by one.

We use the following notation to specify a synchronization code (at least, to specify the family of synchronization codes we focus on). A synchronization code is specified as a sequence taken from the alphabet $\Sigma \cup \{\_\}$. The special character `_` acts as a wildcard. We say that two symbols *match* each other if they are identical or if at least one of them is a wildcard. Sequence A matches sequence B if $|A| = |B|$ and if the character a_i matches the character b_i for all $0 \leq i < |A|$. For instance, the sequence `a _ b _ _ c` matches the sequence

[2] A complete name for the operation ought to be the *rotation to the left by one*. However, this name would be unnecessarily long as we never consider rotations to the right.

`a b _ c _ c`. Let C be a synchronization code and d be the number of wildcards in C. We say that a *clear-text* sequence D of d symbols can be encoded using a synchronization code C to obtain a *synchronized* sequence D_C. One obtains D_C by substituting the first wildcard in C by d_0, the second wildcard in C by d_1, and so on. When $|D|$ is a multiple of d, the successive d-symbol blocks of D are encoded using the same process. For instance, by encoding the original sequence $D =$ `0111` using the synchronization code $C =$ `__110`, we obtain the encoded sequence $D_C = \underline{\texttt{01}}\texttt{110}\underline{\texttt{11}}\texttt{110}$ (for the sake of clarity, the data symbols are underlined).

Here is a list of considerations about synchronization codes.

– **Existence of separate channels for control and data.** In certain transmission configurations, there are two channels available, one for data and one for control, while in other configurations, there is only one channel in which both data and control symbols are transmitted. When there is a single channel and the distinction is possible, we refer to the symbols dedicated to control as *control symbols* and to the symbols dedicated to the transmission of pure data as *data symbols* or *payload symbols.*
– **Transmission through time and/or space.** Transmission through space refers to transmission in the usual sense, i.e. from one point to the other. Transmission through time refers to *storage.* That is, writing symbols on a storage device can be seen as the emission of the symbols and later reading the symbols from the device, as the reception of the symbols. For example, in the case of a hard disk, the emitter and the receiver are the same RW head, but at two different points in time.
– **Presence of a feedback link.** The synchronization mechanism may have access to a feedback link. For example, the TCP/IP protocol uses feedback to control the integrity of the transmission process [10]. On the other hand, a hard disk RW head, when reading a track, might read something that it has written long ago. In a sense, when it reads, the RW completes a transmission through time. If it discovers a synchronization error, the RW head has no means to ask itself to reemit the symbols.
– **Size of the alphabet.** The considered alphabet is either binary or larger.
– **Strength of the synchronization.** A synchronization code provides either hard or soft synchronization. By *hard* synchronization, we mean that, whenever an *a priori* ill-synchronized receiver obtains a sufficient number of error-free symbols, it is guaranteed to be able to recover synchronization. We also say *reliable* synchronization. *Soft* synchronization means that, in the worst case, it might remain impossible to identify the position inside of the data with certainty, even after receiving an arbitrary number of error-free symbols.
– **Blockiness of the synchronization.** A synchronization code works either in a blockwise or in a continuous fashion.
– **Size of the blocks.** If a synchronization code works in a blockwise fashion, it handles either fixed-size or variable-size blocks.
– **Variability of the control-symbol pattern.** A synchronization code uses either a fixed or a variable pattern of control symbols. For example, `__011`

inserts three constant bits. A variable pattern would use three functions to determine the value of the three control bits, based on the data bits. (Note that a variable pattern is not representable using our notation.) Strictly speaking, a fixed pattern is a special case of a variable pattern.

- **Position of the control bits.** If it works in a blockwise fashion, a synchronization code either groups the control symbols in a header or intersperses them among the data symbols.
- **Invariance of the payload symbols.** If it works in a blockwise fashion, a synchronization code either alters or preserves the payload symbols. A typical case where a synchronization code alters the payload data is when there is a header that contains a unique signature that marks the beginning of the block. The synchronization mechanism modifies the payload data to ensure that the signature does not appear by accident inside of the payload part (see, e.g., [5]).

Let us give a succinct description of the synchronization codes that we consider in this paper. Our codes are designed for a single control-and-data channel, with no access to a feedback link, for the binary alphabet $\{\texttt{0}, \texttt{1}\}$. They provide hard synchronization in a blockwise fashion where the blocks have a fixed size before and after the encoding process, they insert a fixed pattern of interspersed control bits, and they keep the original payload bits unchanged.

2.3 The Considered Synchronization Codes

Let us first define the notion of *phase*. Given that we consider transmission in a blockwise fashion, with fixed-size blocks, each bit that is transmitted has a definite position relative to the beginning of the block in which it appears. This is the *phase* of the bit. If one reads a stream of blocks starting at some unknown position—not necessarily at the beginning of a block—one might be interested in identifying the phase of the bits one reads. Indeed, identifying the phase allows one to recover synchronization. Synchronization codes are intended to provide the means to identify the phase. We number the phases starting at zero. The maximum phase is the block size minus one. More generally, the *phase* of a subsequence is the phase of its first bit. If A is a sequence made of one or more blocks and the block size is s, then rotation A^i has phase $i \bmod s$, for any i.

A (d, k, n)*-synchronization code* C is a blockwise code that transforms a clear-text block of d bits into a synchronized block of $(d + k)$ bits. C is denoted by a sequence of symbols taken from the alphabet $\{\texttt{0}, \texttt{1}, \_\}$ where $\texttt{0}$ and $\texttt{1}$ denote control bits and $\_$ denotes a data bit. A (d, k, n)-synchronization code is characterized by its three parameters:

- d is the number of data bits;
- k is the number of control bits; and
- n is the *reliability*, i.e. the number of bits that need to be read, in the worst case, to identify the phase of the sequence.

Table 1. An $(8, 10, 9)$-synchronization code published by [8].

_	_	_	_	0	0	0	1	1	_	_	_	_	0	1	0	1	1
_	_	_	0	0	0	1	1	_	_	_	_	0	1	0	1	1	_
_	_	0	0	0	1	1	_	_	_	_	0	1	0	1	1	_	_
_	0	0	0	1	1	_	_	_	_	0	1	0	1	1	_	_	_
0	0	0	1	1	_	_	_	_	0	1	0	1	1	_	_	_	_
0	0	1	1	_	_	_	_	0	1	0	1	1	_	_	_	_	0
0	1	1	_	_	_	_	0	1	0	1	1	_	_	_	_	0	0
1	1	_	_	_	_	0	1	0	1	1	_	_	_	_	0	0	0
1	_	_	_	_	0	1	0	1	1	_	_	_	_	0	0	0	1
_	_	_	_	0	1	0	1	1	_	_	_	_	0	0	0	1	1
_	_	_	0	1	0	1	1	_	_	_	_	0	0	0	1	1	_
_	_	0	1	0	1	1	_	_	_	_	0	0	0	1	1	_	_
_	0	1	0	1	1	_	_	_	_	0	0	0	1	1	_	_	_
0	1	0	1	1	_	_	_	_	0	0	0	1	1	_	_	_	_
1	0	1	1	_	_	_	_	0	0	0	1	1	_	_	_	_	0
0	1	1	_	_	_	_	0	0	0	1	1	_	_	_	_	0	1
1	1	_	_	_	_	0	0	0	1	1	_	_	_	_	0	1	0
1	_	_	_	_	0	0	0	1	1	_	_	_	_	0	1	0	1

Definition 1 (Synchronization code). *C is a* (d, k, n)-synchronization code *iff it is a sequence drawn from the alphabet* $\{\mathit{0}, \mathit{1}, \_\}$*, it has length* $d+k$*, it contains exactly d wildcards, and the subsequence* $C^i[0..n-1]$ *matches the subsequence* $C^j[0..n-1]$ *only if* $i = j \pmod{d+k}$.

Table 1 presents an example of an $(8, 10, 9)$-synchronization code and illustrates why it is 9-reliable. The code appears on the first row. The other rows are mere rotations of the first row. The reader may verify that, whenever one selects two *distinct* rows of the matrix, one is guaranteed to find a mismatch between two control bits inside of the first 9 columns. Note that one cannot rely on the data bits to cause bit mismatches because data bits are completely out of the control of the synchronization code.

We prove a property of (d, k, n)-synchronization codes that we will reuse later.

Lemma 1. *In a (d, k, n)-synchronization code, the relation $k \leq 2^n - d$ holds.*

Proof. Suppose that one starts reading a stream of bits from an unknown position in that stream. Since the stream is encoded with a n-reliable synchronization code, it is sufficient to read the next n bits in the stream to find the phase in a sure way. There are at most $d + k$ phases and these n bits form one of the 2^n possible sequence of n bits. Each of these sequences can be associated to at most one phase and all phases are covered by at least one sequence which implies $d + k \leq 2^n$ and thus $k \leq 2^n - d$. □

Table 2. Number of occurrences of each substring of 01010000.

Length	**Substrings**							
0	8×ϵ							
1	6×0						2×1	
2	4×00				2×01		2×10	
3	3×000			1×001	2×010		1×100	1×101
4	2×0000		1×0001	1×0010	1×0100	1×0101	1×1000	1×1010
5	1×00000	1×00001	1×00010	1×00101	1×01000	1×01010	1×10000	1×10100
6	1×000001	1×000010	1×000101	1×001010	1×010000	1×010100	1×100000	1×101000
7	1×0000010	1×0000101	1×0001010	1×0010100	1×0100000	1×0101000	1×1000001	1×1010000
8	1×00000101	1×00001010	1×00010100	1×00101000	1×01000001	1×01010000	1×10000010	1×10100000

3 Compression by Substring Enumeration

Dubé and Beaudoin [7] introduced a lossless data compression technique called Compression by Substring Enumeration (CSE). CSE compresses by transmitting to the decompressor in a half-explicit way the number C_w of occurrences of every possible substring w of bits; i.e. for every $w \in \{0,1\}^*$. Table 2 displays the information that needs to be transmitted, in a compressed form, for the sequence 01010000. For example: entry '6×0' indicates that $C_0 = 6$; entry '1×01010000' indicates that $C_{01010000} = 1$; entry '8×ϵ' means that the data is 8-bit long; and the absence of entry for substring 11 means that $C_{11} = 0$. CSE considers the data to be circular. Consequently, we have $C_{00} = 4$, not $C_{00} = 3$, where the fourth occurrence is the one that wraps from the end to the beginning of the data sequence. We say that CSE describes the numbers of occurrences in a *half-explicit* way because the numbers are not transmitted explicitly in a row-by-row fashion. Rather, relations between the numbers of occurrences of different substrings are exploited.

As part of the processing of each substring w, there is the crucial step of predicting the value of the bits that surround all of the occurrences of w. More precisely, for each w that occurs in the original file, CSE predicts C_{0w0}. This prediction is not made from scratch: it is based on the already known values C_w, C_{0w}, C_{1w}, C_{w0}, and C_{w1}. Moreover, it is sufficient for CSE to explicitly describe C_{0w0} as the three remaining unknown values C_{0w1}, C_{1w0}, and C_{1w1} can be deduced from the known ones. Even if it is still in its earliest developments, CSE is already fairly competitive with more mature, well known techniques such as dictionary-based compression [11, 12], prediction by partial matching [13, 14], the Burrows-Wheeler transform [15], and compression using anti-dictionaries [16]. The reader might wonder: what is the link between CSE and synchronization codes?

CSE is an inherently bit-oriented technique. Yet, all the benchmark files (from the Calgary Corpus [17]) that were used are byte-oriented. The conversion of the files into bit sequences is trivial: each byte is simply viewed as a string of 8 bits. However, this change in the point of view of the data is not just cosmetic. It interferes with the prediction steps performed by CSE. Indeed, let us consider what happens when CSE predicts the neighbors of substring `100`. We illustrate the problem using the following hypothetical excerpt from a benchmark file:

$$\ldots\,|\,\mathtt{0\;0\;\underline{1\;0\;0}\;0\;0\;1}\,|\,\mathtt{0\;1\;0\;1\;0\;1\;1\;\underline{1\,|\,0\;0}\;1\;0\;1\;0\;1\;0}\,|\,\ldots$$

in which the two occurrences of `100` have been underlined. The light-grey vertical bars delimit the frontiers between the original bytes but CSE is completely unaware of them. We say that the individual bits in some byte are located at phases 0, . . . , 6, or 7, depending on their position relative to a byte frontier. For instance, the first occurrence of `100` in the excerpt is at phase 2 while the second one is at phase 7. Statistically, it is likely that the '`100`' substrings located at a certain phase have different neighboring bits than the '`100`' substrings located at another phase. Since CSE is unaware of the phase, it makes predictions about the neighboring bits of all the occurrences of `100` at once, without regard to their phases. It has been empirically observed that this phase unawareness incurs some penalty for CSE.

Given this weakness of CSE, the authors had two choices: either adapt CSE to effectively deal with bytes or use a trick to compensate for phase unawareness. The second option appeared to be easier and it directly leads to the family of synchronization codes that we focus on here [7, 8]. Despite the fact that synchronizing original data expands it, the codes in this family are exactly the kind of patterns that CSE is very proficient at detecting and exploiting. Consequently, CSE does not suffer too much from "learning" the newly inserted synchronization code. Moreover, the newly acquired artificial phase awareness leads to better compression rates. We refer the reader to Section 7.

4 A Constraint Model

We present a model to compute a (d, k, n)-synchronization code. The model is built around $2k$ variables that are the positions and values of the control bits in the synchronization code C. Let P_i be the position of the i^{th} control bit and V_i be its corresponding value. Note that we will frequently have to compute some value modulo $(d+k)$, especially after adding two values together or subtracting one value from another. From now on, we write $x \oplus y$ as a shorthand for $(x + y) \bmod (d + k)$ and $x \ominus y$ as a shorthand for $(x - y) \bmod (d + k)$.

We have the following constraints.

$$P_i \in \{0, 1, \ldots, d + k - 1\} \qquad \forall\, 0 \leq i < k \tag{1}$$

$$V_i \in \{0, 1\} \qquad \forall\, 0 \leq i < k \tag{2}$$

To break symmetries, we assume that the control bits are enumerated in the same order as they appear in the sequence.

$$P_{i-1} < P_i \qquad \forall\, 0 < i < k \tag{3}$$

From Definition 1, we know that, for any $i \neq j \pmod{d+k}$, there exists at least one control bit in $C^i[1..n]$ that does not match a control bit in $C^j[1..n]$. Let the mismatching bit in $C^i[1..n]$ be the a^{th} control bit in the synchronization code and the corresponding mismatching bit in $C^j[1..n]$ be the b^{th} control bit in the synchronization code. Since the a^{th} bit and the b^{th} bit are aligned when respectively shifted by i and j bits, we know that $P_a - P_b = i \ominus j$. Moreover, since the a^{th} bit occurs in $C^i[0..n-1]$, its position in the code C is therefore in $\{i, i \oplus 1, \ldots, i \oplus (n-1)\}$. Finally, these two bits exist for any $i \neq j \pmod{d+k}$. We therefore create two variables for each $0 \leq i < j < d+k$.

$$A_{i,j} \in \{0, 1, \ldots, k-1\} \qquad \forall\, 0 \leq i < j < d+k$$
$$B_{i,j} \in \{0, 1, \ldots, k-1\} \qquad \forall\, 0 \leq i < j < d+k$$

Let $P^A_{i,j}$ be the position of the a^{th} bit in the synchronization code.

$$P^A_{i,j} \in \{i, i \oplus 1, \ldots, i \oplus (n-1)\} \qquad \forall\, 0 \leq i < j < d+k$$

Let $P^B_{i,j}$ be the position of the b^{th} bit in the synchronization code.

$$P^B_{i,j} \in \{j, j \oplus 1, \ldots, j \oplus (n-1)\} \qquad \forall\, 0 \leq i < j < d+k$$

Similarly, we define $V^A_{i,j}$ and $V^B_{i,j}$ to be the values of these control bits.

$$V^A_{i,j} \in \{0, 1\} \qquad \forall\, 0 \leq i < j < d+k$$
$$V^B_{i,j} \in \{0, 1\} \qquad \forall\, 0 \leq i < j < d+k$$

These three constraints bind the variables $A_{i,j}$, $B_{i,j}$, $P^A_{i,j}$, and $P^B_{i,j}$ together. The constraint $\textsc{Element}([X_0, \ldots, X_{n-1}], Y, Z)$ ensures that $X_Y = Z$.

$$\textsc{Element}([P_0, \ldots, P_{k-1}], A_{i,j}, P^A_{i,j}) \qquad \forall\, 0 \leq i < j < d+k$$
$$\textsc{Element}([P_0, \ldots, P_{k-1}], B_{i,j}, P^B_{i,j}) \qquad \forall\, 0 \leq i < j < d+k$$
$$P^B_{i,j} = P^A_{i,j} \oplus (j-i) \qquad \forall\, 0 \leq i < j < d+k$$

Finally, since the a^{th} bit does not match the b^{th} bit, their characters must be different.

$$\textsc{Element}([V_0, \ldots, V_{k-1}], A_{i,j}, V^A_{i,j}) \qquad \forall\, 0 \leq i < j < d+k$$
$$\textsc{Element}([V_0, \ldots, V_{k-1}], B_{i,j}, V^B_{i,j}) \qquad \forall\, 0 \leq i < j < d+k$$
$$V^A_{i,j} \neq V^B_{i,j} \qquad \forall\, 0 \leq i < j < d+k$$

This model has a total of $O(k^2 + d^2)$ variables and $O(k^2 + d^2)$ constraints. The cardinality of the domains is bounded by $\max(n, k)$ values.

We show in Section 6 how to further break symmetries.

5 A Pseudo-Boolean Model

The second model we present uses a different approach. Since the synchronization code $c_0 c_1 \dots c_{d+k-1}$ has $d+k$ characters taken from $\{0, 1, \_\}$, we declare two binary variables for each of these characters. The binary variable K_i indicates whether the i^{th} character is a control bit. If the i^{th} bit is a control bit, the variable V_i indicates whether it is a zero or a one.

$$\begin{aligned} K_i &\in \{0,1\} & \forall\, 0 \leq i < d+k \\ V_i &\in \{0,1\} & \forall\, 0 \leq i < d+k \end{aligned}$$

We fix the number of control bits to be k.

$$\sum_{i=0}^{d+k-1} K_i = k$$

We declare a new binary variable Y_i^g for $0 \leq i < d+k$ and $1 \leq g < d+k$. When this variable is equal to one, it implies that the bits c_i and $c_{i \oplus g}$ are distinct control bits, i.e. $Y_i^g = 1 \Rightarrow K_i = K_{i\oplus g} = 1 \wedge V_i \neq V_{i \oplus g}$. We encode this implication with these constraints.

$$\begin{aligned} Y_i^g &\in \{0,1\} & \forall\, 0 \leq i,j < d+k \\ Y_i^g &\leq K_i & \forall\, 0 \leq i,j < d+k \\ Y_i^g &\leq K_{i\oplus g} & \forall\, 0 \leq i,j < d+k \\ (1 - Y_i^g) + V_i + V_{i\oplus g} &\geq 1 & \forall\, 0 \leq i,j < d+k \\ (1 - Y_i^g) + (1 - V_i) + (1 - V_{i\oplus g}) &\geq 1 & \forall\, 0 \leq i,j < d+k \end{aligned}$$

Referring to Definition 1, we make sure that there is a control bit in $C^i[0..n-1]$ that differs from a control bit in $C^j[0..n-1]$. This is ensured by this constraint.

$$\sum_{p \in \{i \oplus a \mid a = 0..n-1\}} Y_p^{j-i} \geq 1 \qquad \forall\, 0 \leq i < j < d+k$$

To break symmetries, we force the variable V_i to be assigned value 0 whenever c_i is not a control bit.

$$K_i \geq V_i \qquad \forall\, 0 \leq i < d+k$$

As for the constraint model of Section 5, this model has a total of $O(k^2 + d^2)$ variables and $O(k^2 + d^2)$ constraints. However, all domains contain two values.

We show in the next section how to break additional symmetries.

6 Symmetries

For any synchronization code C, there exist several other codes that are equivalent. For instance, all zeros in C can be changed to ones and all ones changed to zeros. This produces a valid code. If C is a synchronization code, the rotation C^i is also a synchronization code for any integer i. Finally, reverting the sequence C such that the first character becomes last and the last character becomes first also produces a valid code.

In order to break symmetries in both models, we force the first two characters of the code to be 0 and 1. Indeed, any synchronization code must have two adjacent bits of different value in order for the subsequences $C^0[0..n-1]$ and $C^1[0..n-1]$ not to match. These bits could be anywhere in the sequence C, but after applying a rotation, one can always make them appear at the first two positions of the sequence.

We also force the number of control bits set to zero to be no fewer than the number of control bits set to one.

In the constraint model of Section 4, we add these constraints.

$$P_0 = 0 \qquad P_1 = 1 \qquad V_0 = 0 \qquad V_1 = 1 \qquad \sum_{i=0}^{k-1} V_i \le \frac{k}{2}$$

In the pseudo-Boolean model of Section 5, we add these constraints

$$K_0 = 1 \qquad K_1 = 1 \qquad V_0 = 0 \qquad V_1 = 1 \qquad \sum_{i=0}^{d+k-1} V_i \le \frac{k}{2}$$

7 Experiments

We implemented the constraint model using Gecode 3.7.3 and solved the pseudo-Boolean model using the solver MiniSat+ [18]. All experiments were conducted on a 2.4 GHz Intel Core i7 machine, with 4 Go, 1333 MHz DDR3 RAM.

We tried Gecode using different predefined branching heuristics. The most efficient method was to choose the variable with the smallest domain size divided by the weighted degree of the variable (`INT_VAR_SIZE_AFC_MIN`) [19] — a variable ordering heuristic that gives a higher priority to variables on which failures occur more often. Gecode was able to solve small instances such as $d = n = 8, k = 15$ in 5 minutes and 18 seconds. However, the solver could not prove that the instance $d = n = 8, k = 14$ is unsatisfiable even after a month of computation.

We tried to solve the pseudo-Boolean model with the solver MiniSat+ using the default configuration. The solver solves the instance $d = n = 8, k = 15$ in 0.31 second and proves the unsatisfiability of the instance $d = n = 8, k = 14$ in 3.77 seconds. We also tried the pseudo-Boolean solver Sat4j [20] but MiniSat+ turned out to be more competitive on most instances[3]. We conclude that

[3] For instance, we showed there is no $(32, 13, 32)$-synchronization code after about 6 days using MiniSat+ while Sat4j did not return any answer after 30 days.

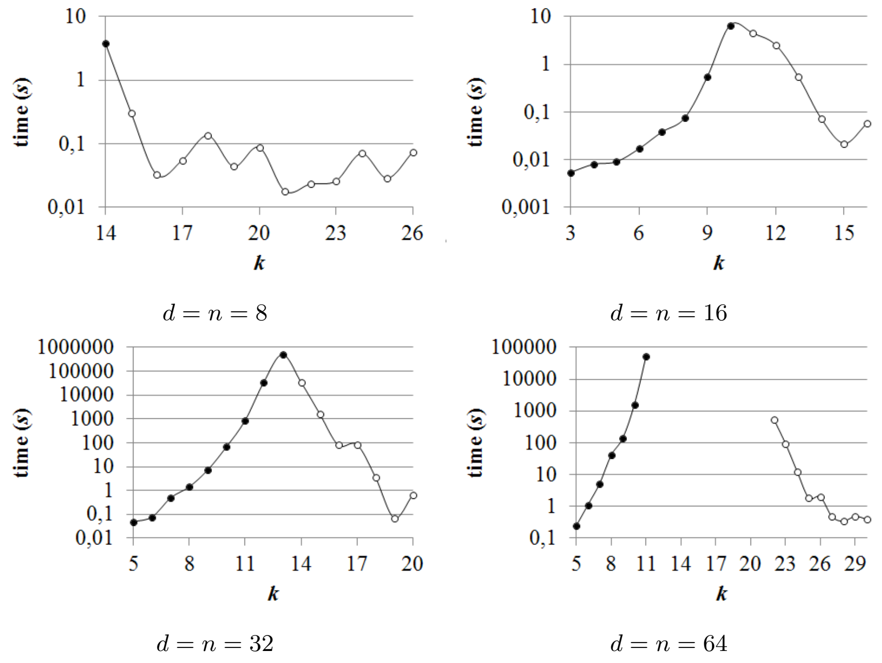

Fig. 1. Time required by MiniSat+ to find a (d, k, n)-synchronization code or prove that no such a code exists. The number of data bits d and the reliability n are fixed, only the number k of control bits vary. A black circle represents an instance for which there is no possible (d, k, n)-synchronization code, and a white circle represents that such code exists.

the combination of the pseudo-Boolean model and the solver MiniSat+ is more efficient than the constraint model.

We first investigate instances with $d = n$ since those are the codes that are the most used. A machine that manipulates data divided into words of d bits is more likely to synchronize the stream of bits after reading $n = d$ bits. Generally, the fewer the synchronization bits, the better. In some cases, a code with additional synchronization bits could perform better for compression purposes.

Figure 1 presents the time needed for MiniSat+ to find a synchronization code or to prove that no such code exists. For (d, k, n)-synchronization codes with $d = n = 8$, $d = n = 16$ and $d = n = 32$ data bits, we proved that the smallest number of control bits is $k = 15$, $k = 11$ and $k = 14$ respectively. For $(64, k, 64)$-synchronization codes, we found a code with $k = 22$ control bits and proved that no such codes exists with $k \leq 11$ bits. We previously solved most of the same instances with the solver Sat4j and proved there is a $(64, 21, 64)$-synchronization code. However, we have not yet proved there is one using MiniSat+, so the result does not show in Figure 1. Whether there exists a $(64, k, 64)$-synchronization

Table 3. Smallest values of k relative to n and d. An entry with ∞ indicates that the solver proved that no such code exists. A blank entry indicates the incapacity of the solver to prove or disprove the existence of the code withing 18000 seconds of computation.

							n						
		2	3	4	5	6	7	8	9	10	11	12	13
	2	∞	∞	6	3	3	3	3	3	3	3	3	3
	3	∞	∞	∞	12	6	4	4	4	4	4	4	4
	4	∞	∞	∞	∞	8	7	4	4	4	4	4	4
d	5	∞	∞	∞	∞		9	7	4	4	4	4	4
	6	∞	∞	∞	∞		8	8	8	7	5	5	5
	7	∞	∞	∞	∞		18	13	9	8	8	5	5
	8	∞	∞	∞	∞		20	15	10	10	8	8	5

Table 4. Synchronization codes used to boost CSE in previous research [8].

n	k	**Synchronization Scheme**
—	0	`_ _ _ _ _ _ _ _`
—	1	`_ _ _ _ _ _ _ _ 0`
—	2	`_ _ _ _ _ _ _ _ 0 1`
—	3	`_ _ _ _ _ _ _ _ 0 1 1`
—	4	`_ _ _ _ _ _ _ _ 0 1 1 1`
13	5	`_ _ _ _ _ _ 0 _ _ 0 1 1 1`
12	8	`_ _ _ 0 _ _ 1 0 0 _ _ 1 _ 1 1 0`
11	8	`_ _ _ 0 _ 0 _ _ 1 1 0 _ _ 1 1 0`
10	10	`_ _ _ _ 0 _ 0 1 1 _ _ _ 0 1 0 0 1 1`
9	10	`_ _ _ _ 0 0 0 1 1 _ _ _ _ 0 1 0 1 1`
8	15	`_ _ _ 0 0 0 1 0 _ _ _ 1 1 1 0 1 _ _ 1 1 0 0 1`
7	20	`1 1 0 1 _ 1 _ 1 1 0 0 _ 0 _ 0 1 0 0 _ 0 _ 1 1 0 0 _ 1 _`

code for $11 < k < 21$ is an open question. MiniSat+ did not succeed to find such codes after 10 hours of computation.

As a second experiment, we try to find the codes with the fewest number of control bits when $d \neq n$. We fix the values of d and n and let MiniSat+ solve the instances with $k = 1, 2, 3, \ldots$ until the solver finds a (d, k, n)-synchronization code. If a (d, k, n)-synchronization code is found, the value of k is written in Table 3. If no (d, k, n)-synchronization code exists for $k \leq 2^n - d$, then Lemma 1 ensures that no synchronization code exists for any value of k and therefore ∞ is written in Table 3. If the solver does not find a code nor prove that it does not exists after 18000 seconds, the entry is left blank.

Table 3 shows that our model is efficient with different instances of d, k, and n. It also validates previously published results obtained using a different methodology [8] and extends it to a wider range of values of d and n.

Table 5. Compression performance obtained while boosting CSE using the synchronization of Table 4, as presented in [8]. Measurements are in bits per character.

Bits/Car.	BWT	PPM	Anti	k=0	k=1	k=2	k=3	k=4	n=13	n=12	n=11	n=10	n=9	n=8	n=7
`bib`	2.07	1.91	2.56	1.98	1.95	1.92	1.92	1.91	1.90	1.89	1.89	1.89	1.89	1.88	1.88
`book1`	2.49	2.40	3.08	2.27	2.26	2.25	2.25	2.25	2.25	2.25	2.25	2.25	2.25	2.29	2.33
`book2`	2.13	2.02	2.81	1.98	1.96	1.95	1.95	1.94	1.94	1.93	1.93	1.93	1.93	1.93	1.95
`geo`	4.45	4.83	6.22	5.35	5.21	4.98	4.81	4.70	4.63	4.58	4.59	4.58	4.58	4.58	4.57
`news`	2.59	2.42	3.42	2.52	2.49	2.46	2.45	2.44	2.43	2.43	2.43	2.43	2.43	2.42	2.42
`obj1`	3.98	4.00	4.87	4.46	4.53	4.43	4.32	4.24	4.17	4.03	4.05	4.02	4.01	4.00	3.99
`obj2`	2.64	2.43	3.61	2.71	2.69	2.59	2.53	2.49	2.47	2.45	2.46	2.45	2.45	2.45	2.44
`paper1`	2.55	2.37	3.17	2.54	2.51	2.48	2.47	2.46	2.44	2.41	2.41	2.41	2.41	2.41	2.41
`paper2`	2.51	2.36	3.14	2.41	2.39	2.38	2.38	2.37	2.36	2.35	2.35	2.34	2.35	2.34	2.34
`paper3`	—	—	—	2.73	2.70	2.69	2.68	2.67	2.65	2.63	2.63	2.63	2.63	2.63	2.63
`paper4`	—	—	—	3.20	3.16	3.13	3.13	3.10	3.07	3.02	3.02	3.02	3.02	3.01	3.01
`paper5`	—	—	—	3.33	3.29	3.27	3.24	3.22	3.19	3.12	3.13	3.12	3.12	3.11	3.10
`paper6`	—	—	—	2.65	2.61	2.58	2.56	2.55	2.52	2.50	2.50	2.49	2.50	2.49	2.49
`pic`	0.83	0.85	1.09	0.77	0.84	0.82	0.82	0.82	0.81	0.81	0.81	0.81	0.81	0.81	0.81
`progc`	2.58	2.40	3.18	2.60	2.58	2.54	2.52	2.50	2.48	2.44	2.44	2.44	2.44	2.44	2.43
`progl`	1.80	1.67	2.24	1.71	1.70	1.69	1.68	1.67	1.66	1.65	1.65	1.65	1.65	1.64	1.64
`progp`	1.79	1.62	2.27	1.78	1.76	1.73	1.71	1.70	1.68	1.66	1.66	1.66	1.66	1.66	1.65
`trans`	1.57	1.45	1.94	1.60	1.58	1.53	1.52	1.50	1.48	1.47	1.47	1.47	1.47	1.47	1.46

Even though this paper addresses the problem of finding synchronization codes, it remains interesting to have an illustration of the effectiveness of the synchronization codes on CSE's performance. Tables 4 and 5 show synchronization codes and compression performance measurements, respectively, presented in previous research by Dubé [8].[4] In Table 4, the n and k parameters are indicated for each code. Certain codes have no associated n; these are unreliable (soft) codes. Apart from the first three columns, the measurements presented in Table 5 correspond to the use of the codes of Table 4. The first three columns present the compression performance of three well-known compression techniques: the Burrows-Wheeler transform, prediction by partial matching, and compression using antidictionaries. The files in the benchmark come from the Calgary Corpus [17]. We refer the reader to the original paper for a complete description of these experiments [8]. One might notice that most of the boosting effect is obtained as soon as the synchronization code is reliable. Still, there exist many applications other than CSE that may benefit from very strong codes.

8 Conclusion

We presented the combinatorial problem of finding a (d, k, n)-synchronization code for CSE compression algorithms. We presented two models for solving the

[4] That previous research has been carried on prior to that presented in this very paper. The synchronization codes that were used have been obtained using various means, some manual, other computational.

problem: a constraint programming model and a pseudo-Boolean model. The pseudo-Boolean model, when solved with MiniSat+, turned out to be more efficient. We were able to produce $(64, k, 64)$-synchronization codes for large instances requiring as few as 21 control bits. We proved that for $(7, k, 7)$-synchronization codes to exist, we need $k \geq 18$, for $(8, k, 8)$-codes, we need $k \geq 15$, for $(16, k, 16)$-codes, we need $k \geq 11$, and for $(32, k, 32)$-synchronization codes, we need $k \geq 14$. These bounds are tight. We also computed the minimum number of control bits required for instances with $2 \leq d \leq 8$ and $2 \leq n \leq 13$.

Acknowledgments The authors would like to thank Julia Gustavsen and Jiye Li for their support while writing this paper. This work is funded by the Natural Sciences and Engineering Research Council of Canada.

References

1. Bruyère, V.: A completion algorithm for codes with bounded synchronization delay. In: Proceedings of the International Colloquium on Automata, Languages and Programming, Bologna, Italy (1997) 87–97
2. Do, L.V., Litovsky, I.: On a family of codes with bounded deciphering delay. In: Proceedings of the International Conference on Developments in Language Theory, Kyoto, Japan (2002) 369–380
3. Golomb, S.W., Gordon, B.: Codes with bounded synchronization delay. Information and Control **8** (1965) 355–372
4. Scholtz, R.A.: Codes with synchronization capability. IEEE Transactions on Information Theory **12** (1966) 135–142
5. van Wijngaarden, A.J., Morita, H.: Partial-prefix synchronizable codes. IEEE Transactions on Information Theory **47** (2001) 1839–1848
6. Stiffler, J.J.: Theory of synchronous communications. Prentice-Hall (1971)
7. Dubé, D.: Using synchronization bits to boost compression by substring enumeration. In: Proceedings of the International Symposium on Information Theory and its Applications, Taichung, Taiwan (2010)
8. Dubé, D.: On the use of stronger synchronization to boost compression by substring enumeration. In: Proceedings of the Data Compression Conference, Snowbird, Utah, USA (2011)
9. Dubé, D., Beaudoin, V.: Lossless data compression via substring enumeration. In: Proceedings of the Data Compression Conference, Snowbird, Utah, USA (2010) 229–238
10. R. Braden, E.: RFC 1122 (1989) `http://tools.ietf.org/html/rfc1122`.
11. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. on Information Theory **23** (1977) 337–342
12. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. on Information Theory **24** (1978) 530–536
13. Cleary, J.G., Witten, I.H.: Data compression using adaptive coding and partial string matching. IEEE Trans. on Communications **32** (1984) 396–402
14. Cleary, J.G., Teahan, W.J.: Unbounded length contexts for PPM. The Computer Journal **40** (1997) 67–75
15. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)

16. Crochemore, M., Navarro, G.: Improved antidictionary based compression. In: Proceedings of the International Conference of the Chilean Computer Science Society. (2002) 7–13
17. Witten, I., Bell, T., Cleary, J.: The Calgary corpus (1987) `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`.
18. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. JSAT **2** (2006) 1–26
19. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI. (2004) 146–150
20. Berre, D.L., Parrain, A.: The sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation **7** (2010) 59–64