File Compression Through Successive Bit Shifted XOR Functions

Rosie Bartlett

Abstract Compression is undoubtedly an exceptionally important part of data storage. Throughout this paper I will be exploring a new compression method that utilises XOR functions and bit shifting to compress data.

Preface

To preface this paper, I would like to first explain the notation used throughout this paper. The method being discussed throught this paper is refered to as SBX, for successive bit shifted XOR compression. SBX uses bit shifting to compress correction keys into shorter keys and bit shift lists, or BSLs, which are simply lists of integers. Since there are multiple BSLs, the BSL related to file n will be refered to as s_n, where the i^{th} value, starting from 1, is refered to as $s_{n,i}$. One of the measures used is in relation to the length of all of the BSLs for a given set of files. In this instance, s refers to all BSLs for the given set of files. Within the scope of this paper, SBX was tested on small amounts of data, and not actual files. For this reason, the term *binary string* is used throughout the paper. This can be seen as equivalent to the term *file*, but in reality it's generation is purely random, unlike a genuine file. The term *pair* is used extensively throughout and refers to a BSL calculation class and key class, and not the actual values contained ithin the classes. For mathematical notation, see table 1.

Compression

SBX uses a parallel iterative process to compress files. This process (see figure the one with the flowchart but a better flowchart) is only run once per set of files and returns the optimal BSL for the respective file. This process is in two parts, the first being preprocessing where files are not considered, and the second being file compression. The entire method is outlined in figure 1. Please note however, that each node is representative of an iterative loop, that when ended returns to the previous node. For the child BSL iteration nodes however, once these have ended, they return to the master pair key iteration node.

Symbol	Description	Type
s_n	Bit shift list for file n	List of integers
k_n	The associated key for file n	Bit string
$D(k_n, s_n)$	Decoding function. See **??**	Function
Ξ	Bitwise XOR shorthand. See **??**	Operation
$<<$	Bit shift left	Operation
I_n	Correction key. See **??**	Bit string
F_n	File n, where n$\neq$ 0	Bit string
F_{0C}	Calculated F_0 for a given key and BSL pairing. This may be an incorrect F_0	Bit string
F_0	Compressed file	Bit string
$L(s)$	Number of items in all bit shift strings	Integer

TABLE 1. *Notation used throughout this paper, in order of occurrence*

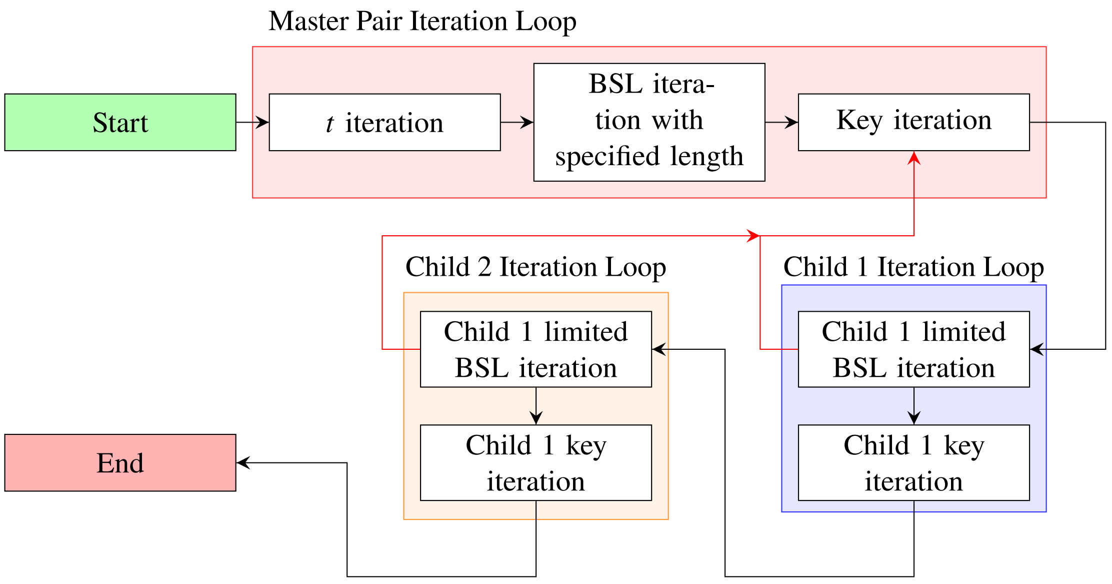

FIGURE 1. *File compression method for three files excluding preprocessing*

Proprocessing

Before any of the files are even considered, a small amount of preprocessing must be performed. This preprocessing consists of populating two empty lists, which will be referred to as s_0 and t. The first list to be populated, s_0, is an exhaustive list of all the possible BSL values. When this list is being populated, teach item added to it is an exhaustive list of BSL values of a specific length, with this length starting at one and ending with a length of l. The second list, t, is all the different combinations of BSL value lengths for all the different files. This is ordered in ascending order where the value being considered is the sum of each value within t.

For example, when n is 2 and l is 3, the first list, s_0 would be `[[[0], [1], [2]], [[0, 1], [0, 2], [1, 2]], [[0, 1, 2]]]`, and the t would be `[[1, 1], [2, 1], [1, 2], [3, 1], [2, 2], [1, 3], [3, 2], [2, 3], [3, 3]]`.

Once this has been completed, the files can now be considered. This consists of the files being allocated to classes that all hold their respective file. The first file is different to the others, as it is held by a master pair class instance which controls all of the compression. Each class contains more data needed for the compression such as the key length, and references to the master pair node class instance for the child nodes, and a list of references to the child nodes for the master pair node.

File Compression

Once the preprocessing has finished, then the compression can begin. The compression is controlled by the master pair node, and begins begins by iterating over each of the items in t. For each of these, it iterates over each of the possible BSL values with a length specified by the first item in the current iteration of t. For each of these iterations, every possible value for the key is iterated over and F_{0c} is calculated, see equation (1), using the current value of the BSL and the current value of the key.

$$F_{0c} = k_1 \oplus s_1 \tag{1}$$

$$I_n = F_n \oplus F_{0c} \tag{2}$$

Once F_{0c} has been calculated, the master pair begins iterating over each of the child nodes. For each child node iteration, a small amount of preprocessing must be performed first to enable checks later on. This preprocessing contains calculating I_n, and then the offset of the first and last non-zero bit in I_n. Once these have been calculated, the child begins iterating over all possible BSL values with a length specified in the current iteration of t in the master class. The index of the child within the master pair class' list of children is the index used in retrieving the current BSL length to iterate over, with the first child accessing the second item in the current iteration of t, the first value being already used for the master pair class.

It is here that the first efficiency check is performed. The first value in the current BSL value has to be less than or equal to the index of the first non-zero bit in I_n, and the last value in the current BSL added to the length of the key must be larger than or equal to the index of the last non-zero bit in I_n. This check ensures the the current BSL is able to produce a non-zero bit in the lowest and highest index required. If this check is passed, then the child node then attempts to calculate a key that, when paired with the respective BSL, would result in I_n. This calculation is performed bitwise, calculating the required bit needed to produce the relevant bit in I_n. This calculation has to be performed bitwise since each successive step requires the previous bits to have been calculated. However, this calculation method is not perfect and errors can be introduced throughout. Because of this, once the key calculation has been performed, the result of the calculation is passed into the key expansion method, along with the respective BSL, the result of which is compared against

I_n. If these are identical then the child returns true to the master pair node, which then continues iterating over the remaining children. If none of the child BSL values could be used, then a new key is generated and the process begins again. Once all children have successfully been iterated over, the master pair and all child nodes save their values and the iterative process is ended. This can be done due to the sorting of t in the preprocessing. Since t was sorted by the sum of each of the elements, then once all children have successfully been iterated over, then there cannot be a shorter values of $L(s)$ for the current set of files, and so iteration is ended.

Expansion

Results

To evaluate the effectiveness of SBX, it was run on several different sets of data, each set having a different batch and file size. The code was run on a three node raspberry pi cluster, appendix time me thinks for detailed specs, and times for each compression have been included. Each set of data contained 1000 pseudo-randomly generated files.

Future Work

Appendix

Raspberry Pi Cluster Specifications

This appendix details the specifications of the Raspberry Pi cluster that SBX was tested upon.

- **Master Node**:
  - Raspberry Pi model B
  - Something else?

Author

Rosie Bartlett is a currently a student and is still developing SBX. She can be reached at abc@xyz.edu. GitHub is also a good way to keep up with the current version of SBX.

Acknowledgements

I would like to thank

Key Words

Keyword one; keyword two; keyword three; keyword four.

Current version: V0.1(March 13, 2021); Date accepted: MMMM DD, YYYY.