Wikant's Compression

An open source algorithm

By Kevin Wikant

Version 1.0

Contents

Introduction

The goal of Wikant's Compression is to take a file encoded into a computer as a stream (a.k.a. an array or sequence) of bits and to map it to a smaller array of bits according to an invertible mapping. Since the mapping is invertible Wikant's Compression is therefore lossless compression.

Throughout this document we will show how the mapping is performed such there is no chance that the inverse mapping could mistakenly map a compressed file back to any more than the single unique uncompressed file that generated it. If there is a possibility that a compressed file could be mistaken for two or more uncompressed files then we will say that the mapping is ambiguous.

We will first need to establish how a file encoded into a modern computer system can be seen as an array of bits.

All computer memory is encoded into an address space. That address space is an array that is typically byte or word addressable. These byte/word addresses thus have a linear ordering. Furthermore, the bits in each byte/word have a linear ordering from most significant bit(MSB) to least significant bit(LSB). Programs and files in memory are organized into sequential memory addresses.

Take the following trivial example encoded into a byte addressable computer system:

Address	**Data**
0x0	01001100
0x1	11100010
0x2	11111000
0x3	11000011

To linearly order this file byte-by-byte the algorithm will do so according to increasing memory addresses. In this example:

$$01001010 \rightarrow 11100010 \rightarrow 11111000 \rightarrow 11000011$$

To linearly order the file bit-by-bit, the algorithm will do so by linearly ordering the bits within each byte from most significant bit(MSB) to least significant bit(LSB). In this example:

$$0 \rightarrow 1 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow \cdots \rightarrow 1 \rightarrow 1 \rightarrow 0 \rightarrow 0 \rightarrow 0$$

Wikant's Compression relies on the methodology described above to linearly order a file into an array of bits. As a consequence it may only be applied to files & programs encoded into consecutive memory addresses. Fortunately, most programming languages offer the ability to read & write files in terms of arrays of bytes.

Since a file will be in the form of an array of bytes rather than an array of bits, in order to read/write individual bits to/from each byte we will need to employ the following techniques:

1. To write individual bits to a byte, generate a byte of value 1 then logical shift it left to the required bit position then perform a bitwise OR with the byte to write a 1 to that bit position while leaving all other bits unchanged.

2. To read individual bits from a byte, generate a byte of value 1 then logical shifting it left to the required bit position then perform a bitwise AND with the byte to mask all other bits to 0. If the resultant byte is nonzero then the bit at the bit position must be 1, otherwise if the resultant byte is 0 then the bit at the bit position must be 0.

In reality the actual implementation of Wikant's Compression for both the mapping & inverse mapping will work for the most part by performing an in-order traversal of the of the array of bytes in order to dynamically generate the result of the mapping as the in order traversal proceeds.

However, in the following sections we will ignore the details of the implementation of Wikant's Compression on an actual computer system. We will not consider how best to optimize the runtime and memory requirements of the algorithm. Rather, we will simply consider a file as an abstract construct of an array of bytes that we can manipulate freely without the restrictions of the actual implementation.

It is significant to note that the reason that Wikant's Compression must linearly order the bits of a file is that it uses the concept of the index of a bit in the array of bits in order to perform its compression.

We will examine how to perform the algorithm of an array of bits. First as a generalization. Then using the example file:

$$01001100111000101111100011000011$$

We call the array of bits that gives a file encoded in its standard form, the "Uncompressed Stream".

The array of bits that algorithm maps the Uncompressed Stream to, is called the corresponding "Compressed Stream".

Division of the Uncompressed Stream into Subsections

Both the mapping & inverse mapping of the uncompressed & compressed streams work on the basis of dividing the array of bits into an array of sections, where each section is a subsequence of the array of bits. Each section is mapped individually then the resulting mapped sections are rejoined in the same linear ordering that they follow in both the uncompressed & compressed streams.

Here we will consider how to break the uncompressed stream into sections according to the following:

1. Starting from the beginning of the file the first section consists of any consecutive 0 bits not preceded by any 1 bits. We refer to this section as the leading 0 bits. It does not necessarily exist in a given file.
2. After the leading 0 bits we will encounter a number of consecutive 1 bits. Any number of consecutive 1 bits followed by any number of consecutive 0 bits is referred to as an internal section.
3. The final section is the one at the end of the uncompressed stream that consists of any consecutive 1 bits not followed by any 0 bits. We refer to this section as the lagging 1 bits. It does not necessarily exist in a given file.

Any given array of bits must be divisible into these sections.

In the case of our example the division would proceed:

$$0 \mid 100 \mid 1100 \mid 111000 \mid 10 \mid 11111000 \mid 110000 \mid 11$$

Where | represents an uncompressed section divider.

You will note that we may generate these internal sections simply by traversing the uncompressed stream bit-by-bit since we may clearly determine where each section:

- Begins: as marked by the bit after the end of the previous uncompressed section.
- Ends: as marked by the first 1 bit encountered that breaks the sequence of consecutive 0 bits.

Mapping the Internal Sections of the Uncompressed Stream

We will begin by considering the internal sections of the uncompressed stream for which Wikant's Compression is designed to for.

For simplicity, in any given internal uncompressed section let:

$$x = number\ of\ 1\ bits$$

$$y = number\ of\ 0\ bits$$

The original idea behind the algorithm was that given an internal section that consists of any x & y we could recode such a section to be represent the values of x & y by the use of index in an array of bits.

In particular, if in the corresponding compressed section we let 0 bits represent null place holders in an array of bits, we could then use 1 bits to flag certain indices in that array. These flagged indices could be used to give the values of x & y. The reason that this has the potential to reduce the size of the file is that we can represent both the values of x & y in an array that needs only be the size of the larger of x & y.

Based on this methodology, Wikant's Compression can be thought of as an indexed based recoding of an array of bits.

However, there are problems that arise when attempting to recode an uncompressed section into its indexed based compressed section format. In particular:

1. We must be able to distinguish which flagged index gives x and which flagged index gives y. The smaller of x & y will appear as the first flagged index and the larger will appear as the second flagged index. In the case that they are equal, we need a single flagged index to give both x & y. This begs the question, for a given compressed section how do we know which of the above is the case for the corresponding uncompressed section?
2. Since the encoding of each compressed section relies on our ability to treat it as an indexed array of bits. We must clearly be able to distinguish where each compressed section ends such that we know where the next compressed section begins. However, the ability to do this comes with the knowledge of how many flagged indices exist in the in the compressed section (either 1 or 2), once we encounter the last flagged index we know is the last that will mark the end of the compressed section.

To deal with the dilemmas described above, it became necessary to introduce at the begging of each compressed section 2 flag bits $b_1 b_2$. These are not to be confused with the idea of flagging an index with a 1 bit to give the values of x & y. These flag bits instead act as possible states each of which tells us about how the values of x & y compare to each other in the corresponding uncompressed section.

Thus the structure of a compressed section consists of 2 flag bits $b_1 b_2$ directly followed by an indexed array of bits where 0 bits represent placeholders and 1 bits are used to flag indices where the values of these indices give the values of x & y.

Furthermore, based on the flag bits of a given compressed section we will be able to determine:

1. How many flagged indices there are and thus where the compressed section ends based on where the final flagged index is.
2. Which flagged index gives which value of x or y.

For reasons that have to do with the fact that there are 2 flag bits, it becomes practical to consider as special cases the sections in which one of x & y is equal to 1.

Let us now consider the specific cases of compressed sections. Each case is based on the flag bits that it requires. Immediately following the flag bits is the indexed encoding of the values of x & y. However there are some things we must note:

1. For cases 1 & 2 the there is no possibility that the index 1 in the indexed encoding is filled since these are covered by cases 3 & 4. Thus this provides us with an additional flag bit b_3 that goes at the position where the index 1 would be. Then the indexed encoding followinb_3g begins with index 2. b_3 allows us to define further special cases.

Case 1 : $b_1 b_2 = 00$ where $x \geq y > 1$

Since both x & y are greater than 1, the first bit of the in indexed section can never be 1. Thus we have a 3rd flag bit available b_3 that immediately follows $b_1 b_2$. Following the third flag bit b_3, the indexed section will begin with a bit position giving index 2. There are 2 sub cases:

Case 1a : $b_3 = 0$ in this case we assume $x \neq y$ and so in the indexed section we should expect two flagged indices, the first being the value of y and the second being the value of x which also marks the end of the compressed section.

Case 1b : $b_3 = 1$ in this case we assume $x = y \neq 1$ and so in the indexed section we should expect a single flagged index that gives both the values of x & y and marks the end of the compressed section.

Case 2 : $b_1 b_2 = 01$ where $y \geq x \geq 1$

Since both x & y are greater than 1, the first bit of the in indexed section can never be 1. Thus we have a 3rd flag bit available b_3 that immediately follows $b_1 b_2$. Following the third flag bit b_3, the indexed section will begin with a bit position giving index 2. There are 2 sub cases:

Case 2a : $b_3 = 0$ in this case we assume $x \neq y$ and so in the indexed section we should expect two flagged indices, the first being the value of x and the second being the value of y which also marks the end of the compressed section.

Case 2b : $b_3 = 1$ in this case we assume $x = y = 1$ and as such the compressed section ends immediately following b_3.

Case 3 : $b_1 b_2 = 11$ where $x = 1$ & $y > 1$

The value of x is always 1 .The indexed section following these flag bits begins with the bit position giving index 2, it consists of a single flagged index that gives the value of y and marks the end of the compressed section.

Case 4 : $b_1 b_2 = 10$ where $y = 1$ & $x > 1$

The value of y is always 1 .The indexed section following these flag bits begins with the bit position giving index 2, it consists of a single flagged index that gives the value of x and marks the end of the compressed section.

These 4 cases cover how to generate all possible compressed sections that correspond to internal uncompressed sections.

Mapping the Leading 0 Bits & Lagging 1 Bits of the Uncompressed Stream

Any leading 0 bits & lagging 1 bits represent a deviation from the standard internal sections for they do not entail both an unbroken sequence of 1 bits followed by an unbroken sequence of 0 bits. Thus we must treat them as special cases.

In the uncompressed stream these sections are found at the beginning and end of the stream respectively. In the compressed stream we will encode the information on both these sections into the beginning as a section that we will call the initial section in the compressed stream.

What we need to consider to generate this initial compressed section is the number of leading 0 bits & the number of lagging 1 bits in the uncompressed stream. In fact, together these leading 0 bits & lagging 1 bits represent the same relevant information that is given in each internal section of the uncompressed stream. However we cannot treat this as a standard uncompressed internal section for the reason that these sections potentially do not exist in the uncompressed stream.

To avoid ambiguity we must ensure that this initial compressed section always exists so that there is no mistaking the first compressed section in the compressed stream for an internal compressed section.

To do this we simply encode the values of the number of leading 0 bits & the number of lagging 1 bits plus 1 into the initial compressed section. So we treat the initial compressed section as a standard internal section where:

$$x = number\ of\ lagging\ 1\ bits + 1$$

$$y = number\ of\ leading\ 0\ bits + 1$$

So if there are no leading 0 bits & lagging 1 bits we will still have $x = y = 1$ ensuring that the initial compressed section always exists.

As a consequence of this in performing the inverse mapping will need to subtract 1 from the values of x & y that are determined from the initial compressed section to get the numbers of leading 0 bits and lagging 1 bits.

Consequence of the Encoding of files into Bytes

There is one more consideration that needs to be made as a consequence of the division of the stream of bits into bytes.

The compressed stream will not necessarily be a size that divides perfectly into bytes. As such at the end of the mapping of the uncompressed stream into the corresponding compressed stream we will have to append some number of 0 bits required to make the compressed stream fill the remaining indices required to fit strictly into bytes.

This works since all compressed sections must end in a 1 bit, so these appended 0 bits used to fill the final byte may not be mistaken for a compressed section since they will not match any possible encoding of a compressed section.

Example Mapping

Lets not take our example and compress it.

$$0 \mid 100 \mid 1100 \mid 111000 \mid 10 \mid 11111000 \mid 110000 \mid 11$$

There leading 0 bits and lagging 1 bits give:

$$x = 2 + 1 = 3$$

$$y = 1 + 1 = 2$$

These values of x & y give case 1a) $x > y > 1$ which gives the flag bits 000 and the indexed section 11 giving the compressed section:

$$00011$$

Uncompressed Section	x & y	Case	Compressed Section
100	$x = 1 \,\&\, y = 2$	3	111
1100	$x = 2 \,\&\, y = 2$	1b	0011
111000	$x = 3 \,\&\, y = 3$	1b	00101
10	$x = 1 \,\&\, y = 1$	2b	011
11111000	$x = 5 \,\&\, y = 3$	1a	0000101
110000	$x = 2 \,\&\, y = 4$	2a	010101

Flag bits are in red.
Indexed section is in blue.

After having compressed each section individually to generate the compressed stream we put the initial section at the beginning and then simply join all corresponding compressed sections in the order that they appear in the uncompressed stream. Thus we have:

$$00011 \mid 111 \mid 0011 \mid 00101 \mid 011 \mid 0000101 \mid 010101$$

When rejoined:

$$00011111001100101011000010101010101$$

And ofcourse due to the encoding of a computers into byte addressable units we will need to divide the compressed stream into bytes:

$$00011111\ \ 00110010\ \ 10110000\ \ 10101010\ 1$$

Since the compressed stream does not fit into bytes, we must append consecutive 0 bits to fill the last byte:

$$00011111\ \ 00110010\ \ 10110000\ \ 10101010\ 10000000$$

You will note how in this case Wikant's Compression actually makes the file larger.

The Inverse Mapping

The inverse mapping should come fairly intuitively with the knowledge of the mapping.

To convert the compressed stream back into its uncompressed stream we must traverse it according to the linear ordering of bits, this is necessary since the only way to identify the end of each compressed section is in considering the flag bits of the section.

Thus the separation of the compressed stream into compressed sections and the inverse mapping of each compressed section back to the corresponding compressed section proceeds according to the following methodology in which we traverse the compressed stream bit-by-bit:

1. Beginning with the very first two bits of the compressed stream giving the flag bits of the initial compressed section proceed as follows,
2. Use the flag bits of each compressed section to determine how many flagged indices are in the indexed section,
3. Using the knowledge of how many flagged indices to expect we can determine where the compressed section ends,
4. The next two bits following the end of a compressed section are the flag bits of the following compressed section, repeat step 2 using them.
5. Eventually there will come a point where we encounter only consecutive 0 bits followed by no 1 bits as a consequence of the encoding into bytes, since these do not represent any possible encoding of a compressed section they may be discarded.

As we get the values of x & y from the flagged indices in each compressed section, the inverse mapping to the corresponding uncompressed is achieved simply by generating x 1 bits followed by y 0 bits. There will need to be special considerations made for the values of the leading 0 bits and lagging 1 bits as given by the initial compressed section.

Analysis & Flaws of Wikant's Compression

We will consider now the cases of where Wikant's Compression is the most effective and the cases where it can actually make a file larger. In particular we will consider the amount of bits that can be reduced or added from a particular uncompressed section when mapped to its corresponding compressed section.

In general an internal uncompressed section has size:

$$x + y$$

Based on which case the uncompressed section represents, the corresponding uncompressed section can have size:

- For cases 1a, 1b, & 2a the size is given by:

$$2 + \max(x, y)$$

If $\min(x, y) = 2$ then the compressed section is equal in size to the uncompressed section.

Otherwise the compressed section is $\min(x, y) - 2$ bits smaller than the uncompressed section. This is where Wikant's Compression shows true prowess, for the uncompressed section that consist of a large number of 1 bits followed by a large number of 0 bits.

- For cases 3 & 4 the size is given by:

$$1 + \max(x, y)$$

The compressed section is always equal in size to the uncompressed section.

- For case 2b the size of the compressed section is always 3 which is always 1 bit larger than the size of the uncompressed section.

These cases differ slightly when considering the section given by the leading 1 bits and lagging 0 bits in which case we must factor in the fact that we are adding 1 to both the values of x & y in generating the initial compressed section.

Final Remarks

At its essence Wikant's Compression can be thought of as an index based re-encoding of array of bits. Its power comes from its ability to greatly reduce compressed sections that consist of a large sequence of 1 bits followed by large sequence of 0 bits.

Wikant's Compression is lossless and it does not rely on eliminating redundancies as most compression algorithms do.

It is important to note that while Wikant's Compression has the potential to reduce the size of a computer file by re-encoding it, it is also possible for it to increase the size of a file. Although this may sound like a terrible flaw in the algorithm, it can proven mathematically using the pigeonhole principle that any compression algorithm that is able to make at least one file of size N smaller it is guaranteed to make at least one file of size N larger. Wikant's Compression is no exception to this rule.

The distinction of how effective Wikant's Compression is, is made based on how well suited the uncompressed sections are for the algorithm. As seen in in the analysis section, we note that different uncompressed sections can have their sizes reduced, equaled, and increased when mapped to their corresponding compressed section. Thus the frequency of each case of uncompressed section in the uncompressed stream is the determining factor on how effective Wikant's Compression will be.

I myself have already envisioned an enhanced way of performing Wikant's Compression not implemented in this version, it is discussed in the section following this one.

Other potential ways to enhance the algorithm could entail:

- Reducing the number of flag bits required, while still maintaining an unambiguous mapping.
- Re-envisioning the index based recoding of the uncompressed sections all together.
- Change how the uncompressed sections are divvied up (i.e. rather than consecutive 1 bits followed by consecutive 0 bits)

My primary issue in these attempted enhancements is that they have led to situations where the inverse mapping becomes ambiguous and can no longer deterministically produce the uncompressed stream that generated it.

Although I believe that the idea of an indexed based recoding is a good one for compression purposes, I do not believe that the implementation I have chosen is the optimal one. There are no doubt many different ways to use the same idea and I encourage readers to attempt to come up with a better algorithm.

Should anyone derive a solution to any of the flaws in Wikant's Compression then please come forward with your information on how to do so. If your contribution assists in improving upon the algorithm in a meaningful way then I will gladly give you due credit by renaming the algorithm to:

Wikant-"Your name" Compression

To the best of my knowledge, I am the first to have envisioned the idea of an indexed based recoding of an array of bits. Please inform me if I am incorrect in saying so.

For any questions, concerns, & comments I can be reached at:
kpwikant@uwaterloo.ca

Enhancement Technique

Consider the fact that there are many different ways we could choose to encode the relevant information on the values of x & y in the corresponding uncompressed section into the flag bits of the compressed section. I have tried many of such encodings before settling on the one described above for I believe it offers in general the best compression.

However, it is apparent that different but similar encodings of the flag bits of the compressed section can be designed to optimize the reduction of certain types of uncompressed sections while having negative impacts on the size of certain other types of uncompressed sections. These differences in encodings will thus have different effects on the same uncompressed stream based on the frequency of each type of uncompressed section within it.

For example, the encoding given above is guaranteed to only increase the size of the uncompressed section 10 by a single bit. However, if this section appears frequently in the uncompressed stream, then it is likely to increase in size. A different encoding of the flag bits, one that I have tried before, will not increase the size of the uncompressed section 10 and so it would be better suited for such an uncompressed stream.

From this knowledge, a potential option for Wikant's Compression would be to have several different encoding types for the flag bits of the compressed sections. The algorithm would attempt each and use only the one with the optimal compression ratio. It could then identify which encoding is used for the compressed stream by using n encoding type bits at the very beginning of the compressed stream where there are 2^n possible encoding types of the flag bits.