CITS2200: Data Structures & Algorithms

Lempel-Ziv Welch Data Compression

Kuraara Forrest 20488223

1 Lempel Ziv Welch

1.1 Summary

The Lempel-Ziv Welch (LZW) compression algorithm is a variation on the LZ-77 algorithm that dynamically builds a dictionary of string patterns previously encountered.

The dictionary D is made of the tuples containing words and numbers which those words are indexed by. For instance, let A be the alphabet of [A – Z], then the initial dictionary would contain 26 entries and their associated index e.g. D = { {A, 1}, {B, 2}, {C, 3} … {Z, 26} }. These code values have a wider bit width than their respective widths in the base alphabet, A – Z require 2^5 bits to hold their binary representation but for any high values beyond 32, the codes need to be stored in larger chunks of space. In order to be effective, the input then needs to be sufficiently large enough, and the size of the dictionary needs to have an upper limit that ensures larger code values are output more frequently than the lower values.

Using the string "WOODCHUCKSCANNOTCHUCKWOOD"

With the base dictionary is encoded as:

W	O	O	D	C	H	U	C	K	S	C	A	N	N	O	T	C	H	U	C	K	W	O	O	D
23	15	15	4	3	8	21	3	11	19	3	1	14	14	15	20	3	8	21	3	11	23	15	15	4

The base dictionary of {A,1} – {Z,26} is then extended out to:

WO	OO	**OD**	DC	**CH**	HU	**UC**	CK	KS	SC	CA	AN	NN	NO	OT	TCH	CHU	UCK	KWO	WOO	OOD
27	28	**29**	30	**31**	32	**33**	34	35	36	37	38	39	40	41	42	43	44	45	46	46

The compressed string then becomes

W	O	O	D	C	H	U	C	K	S	C	A	N	N	O	T	**CH**	**UC**	K	**WO**	**OD**
23	15	15	4	3	8	21	3	11	19	3	1	14	14	15	20	**31**	**32**	11	**27**	**29**

The highlighted cells denote the re-use of the extended dictionary, the coloured cells indicate when these entries are printed out, while dark grey indicates new entries.

During each step of the compression process, as a new word is added to the dictionary, the code value of the previous word is then output. After each subsequent encounter of these defined patterns, the code is output, and a new entry is made that expands the old entry to include the next character encountered.

One of the advantages of the LZW algorithm is the fact that it does not require any pre-processing of the file. In algorithms such as Huffman and LZ77/78, the probability distributions must be known in advance. So generating the code dictionary requires processing the file at least once, transmitting the decode table followed by the compressed payload.

Coping with changes in the frequency of known patterns is simpler to deal with in LZW. Should the occurrence of know word sequences fall below a certain threshold once the dictionary is full, the dictionary can then be reset, and a clear code written to the output stream. *I had tried to implement this feature however I couldn't quite figure out how this should be calculated.*

The reasons for choosing LZW over the other compression algorithms were:

- It was conceptually a lot easier to understand;
- Has fewer steps than algorithms like adaptive Huffman Encoding or LZ77/78;
- Overcoming the issue of packing variable width bytes would require more effort than the effectiveness of the algorithms would achieve.
- The fact that it essentially decompresses itself with no reliance on the original dictionary used to encode it seemed most interesting;

2 Description of the algorithm used

2.1 Pseudo-code

2.1.1 Compression

```
Initial Dictionary D = { 0, 1, ... 255 }
w = null
while( k == next byte)
      if wk ∈ D
            w = wk
      else
            add wk to D
            output code for w
            w = k
output code for w
```

2.1.2 Decompression

```
Initial Dictionary D = { 0, 1, ... 255 }
k = next byte
output k
w = k
entry = { ∅ }
while( k = next byte )
      if k ∉ D
            entry = w + first letter of w
      else
            entry = D{k}

      output entry
      add (w + entry(0) ) to D
      w = k
```

3 Data Structures & Complexity Analysis

For LZW to be effective, the input must meet the following conditions:

1. Be sufficiently large enough that values in the extended dictionary are used and;
2. Have a relatively even distribution of repeating patterns. Generally speaking, the compressed output size is inversely proportional to time taken.

The dictionary size and byte packing determine the maximum file size in the worst case compression scenario. The maximum code value is therefor going to have a width of 12 bits which is 1.5 bytes. It would stand to reason that this would limit the absolute worst case performance to an output 1.5x larger than the input.

Elements will be added to the dictionary at a rate of O(n) and in this implementation, will have an upper bound of 2^{12} = 4096.

3.1 Data structures used

A **TreeMap** was the ideal choice for the compression dictionary because lookups and insertions are based on size comparisons rather than a **HashMap** due to the potentially unpredictable performance in the event of hash collisions. A Binary Tree based dictionary offers a guaranteed maximum of 12 comparisons per lookup and insertion. As the input size and hence, number of lookups tends towards infinity, this can assumed to be a constant time operation.

The decoding step is a simpler affair which allows the use of an **Array List**. It offers the functionality of a LinkedList Dequeue structure but with the constant time

performance of an array. The costly resize operation is not an issue since the list can immediately be initialized to the maximum size of the dictionary.

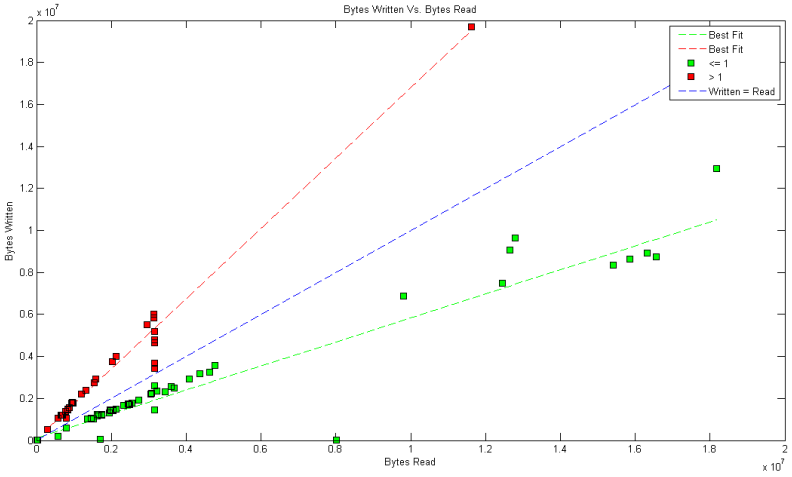
Word sequences were managed using the **StringBuffer**, this is to take advantage of the fact that Strings are simply collections of chars, which are easily type-cast back and forth with ints.

The ReadBuffer and WriteBuffer classes are both low level stacks that use bit shifting to serialize the data for compression / decompression.

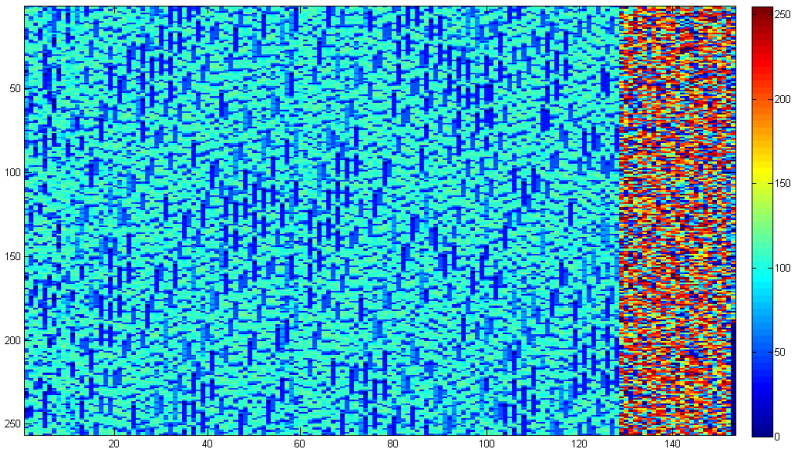
4 Experimental analysis and efficiency

4.1 Compression achieved

Compressed data of order less than or equal to 1 was generally within the range of 45% - 85%. There were a few outliers where large files with no variation at all could go as low as 1%, this however was used to test the worst case time taken.



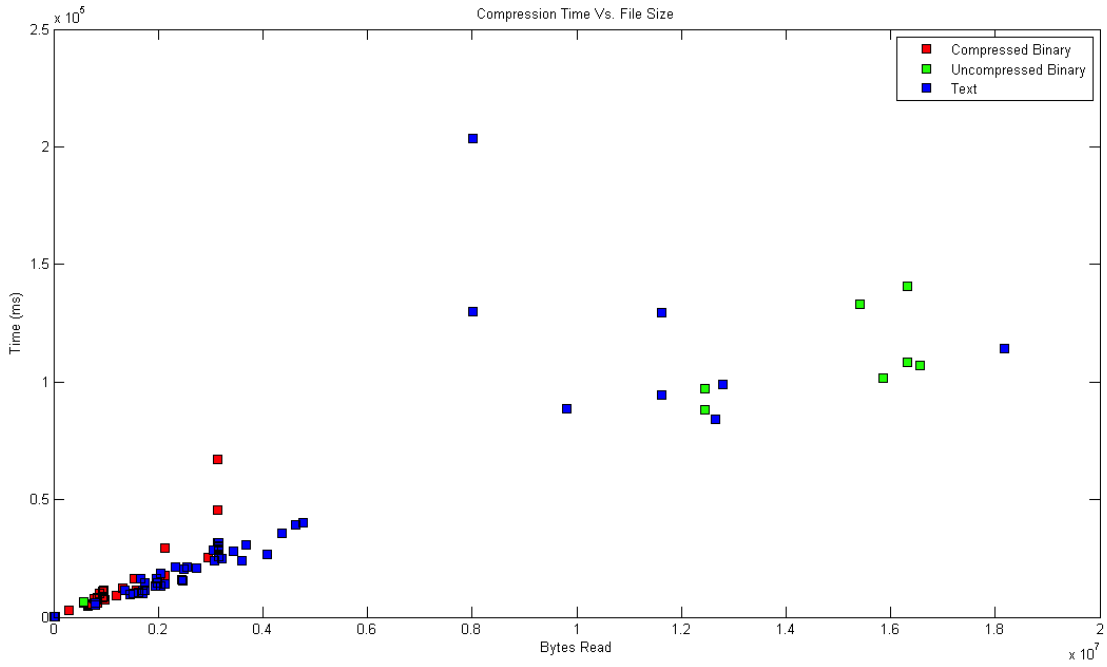
The following figure demonstrates the density and overall sizes of a compressed vs uncompressed file this implementation was used on. On the left (taking up most of the plot) is the uncompressed file, with relatively low byte values. This is in stark contrast to the compressed file occupying a fraction of the space, but with higher values overall.

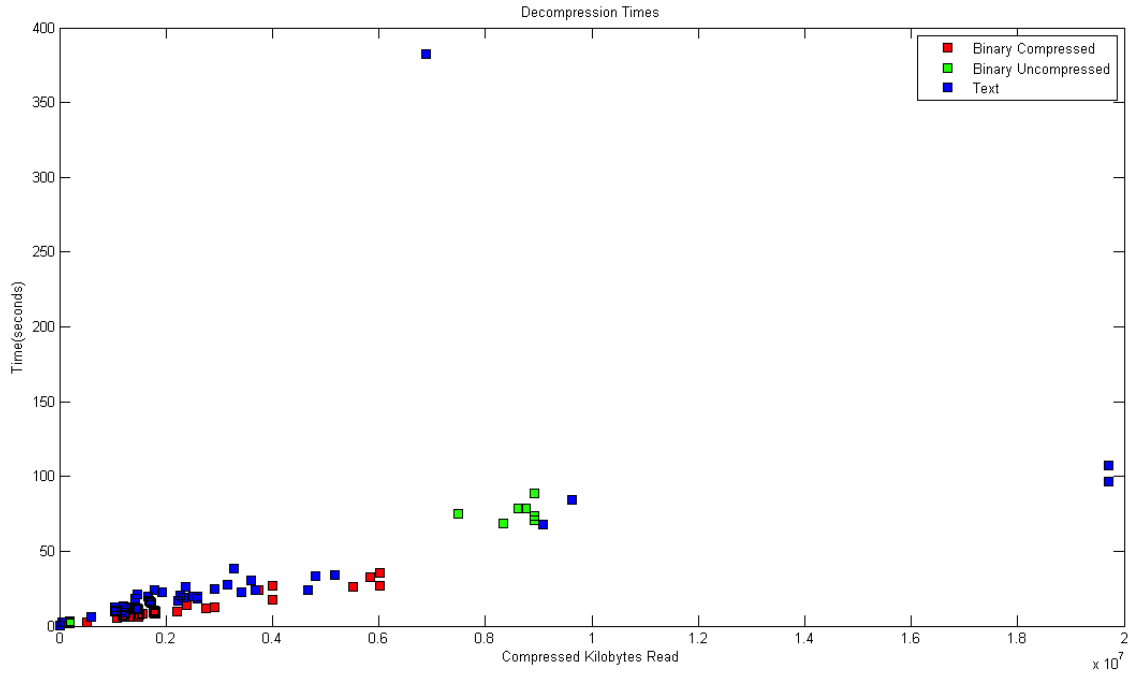


4.2 Compression and Decompression Times

In both instances the time taken was fairly linear in growth with respect to bytes read.

Text files had a tendency to compress exceptionally well followed by binary data (bitmap, .x direct-x model files). Files that compressed poorly achieved better times as the dictionary would fill up quicker with many small entries that can be compared in a shorter period of time.




Decompression Times
Binary Compressed
Binary Uncompressed
Text
Time(seconds)
Compressed Kilobytes Read
x 10^7
0
50
100
150
200
250
300
350
400
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2