The size of

the dictionary is usually limited because of memory and

compression time limitations, and therefore the dictio-

nary stores only the newer part of the string. This type

of dictionary is called a

sliding dictionary

.

To compress a string well, we have to find the

longest match string in the dictionary. It is also impor-

tant to find the nearest one among the longest match

strings because the nearest one is encoded in fewer bits.

The LZ77 compression using the sliding dictionary

can be done in linear time

However, the algorithm

requires huge memory and it is not fast in practice. An-

other problem is that it cannot find the nearest string

in the dictionary.

The window size can be increased, but this will cause two losses. One is that the algorithm

processing time will increase for it should perform once character matching with prior

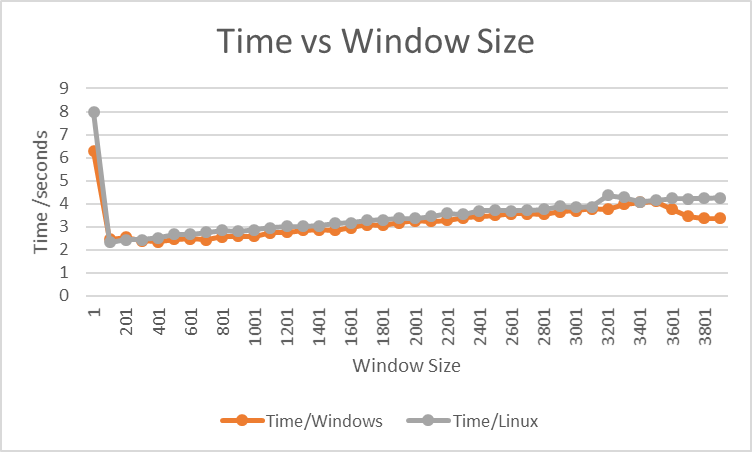
cache for each location of sliding window. The second is the field should be longer to

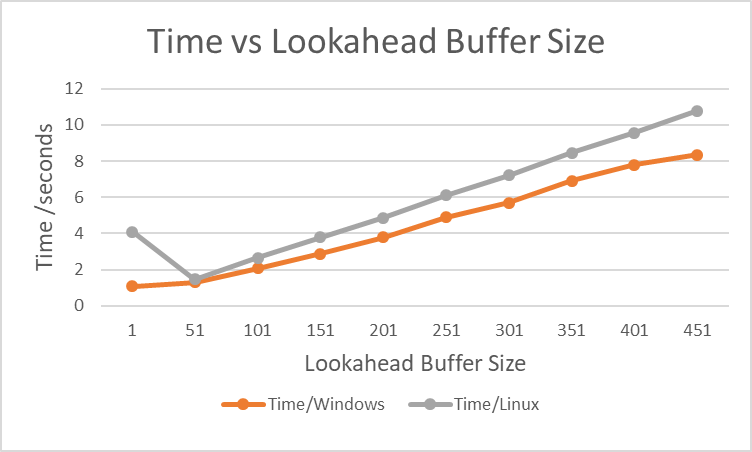
permit longer jump.

The running time of the compression algorithm is linear, that is by using a window and a lookahead buffer the data can be compressed in linear time. The size of the window is usually limited due to lack of memory and so the window stores only the newer part of the string. As we move from the start of the string to the end, the window and the buffer move as the current position in the string changes. In my implementation the function that takes the most time is the function to find the longest match in the string. This is because it consists of many if-else statements and a for loop as well. Different experiments were carried out as can be seen from the graphs below.

**Lempel-Ziv coding (LZ77)**

1. ***Running time of the encoder (compression)***

*******File name:* **22.txt** *File size:* ***38.1KB***

******

**Max** time Linux: 10.789 sec

**Min** time Linux: 1.467 sec

**Average:** 5.899 sec

**Max** time Windows: 8.360 sec

**Min** time Windows: 1.084 sec

**Average:** 4.475 sec

**Max** time Windows: 7.977 sec

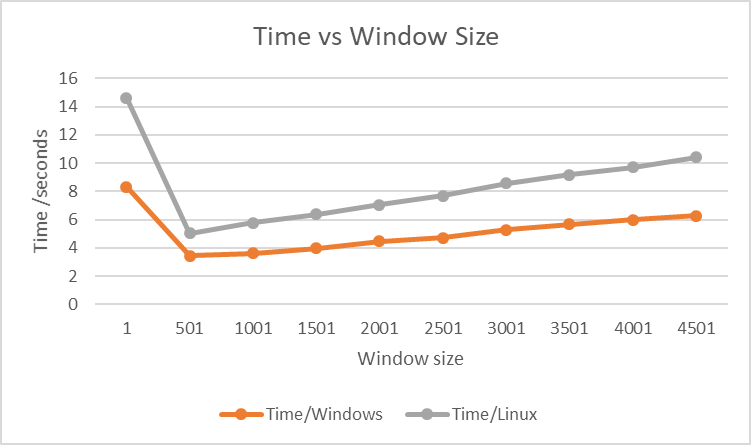
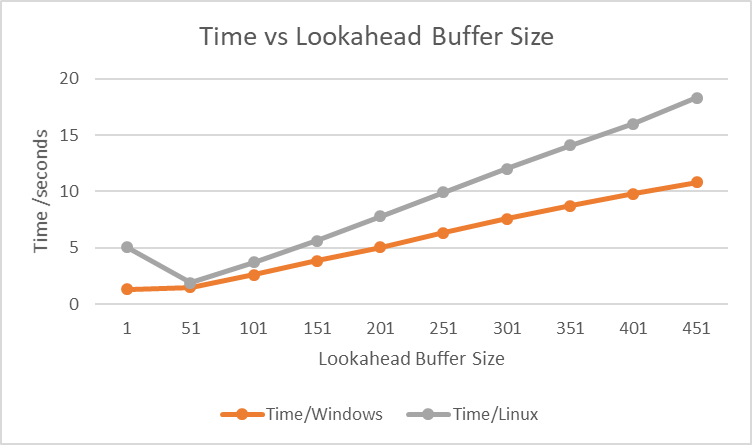
**Min** time Windows: 2.323 sec

**Average:** 3.512 sec

**Max** time Windows: 6.281 sec

**Min** time Windows: 2.353 sec

**Average:** 3.227 sec

*******File name:* **memetics.txt** *File size:* ***46.5KB***

**Max** time Windows: 8.340 sec

**Min** time Windows: 3.425 sec

**Average:** 5.168 sec

**Max** time Linux: 14.621 sec

**Min** time Linux: 5.036 sec

**Average:** 8.447 sec

**Max** time Windows: 10.793 sec

**Min** time Windows: 1.350 sec

**Average:** 5.754 sec

**Max** time Linux: 18.287 sec

**Min** time Linux: 1.897 sec

**Average:** 9.438 sec

*File name:* *File size:*

The experiments that I carried out were done by having a constant window size and changed the buffer size at constant intervals and then vice versa.

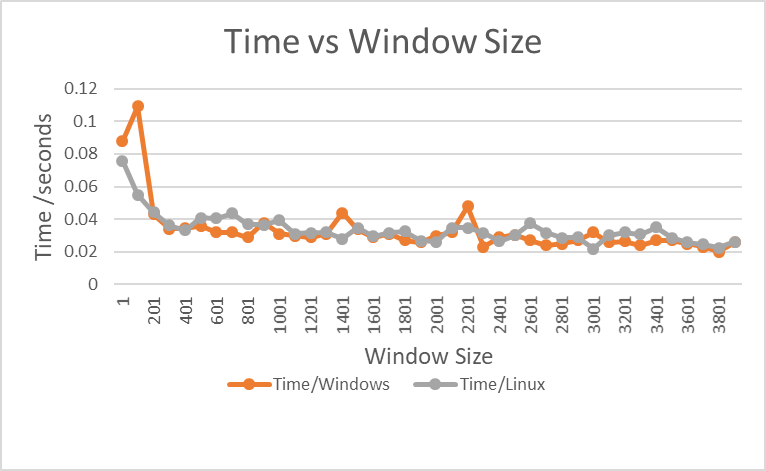
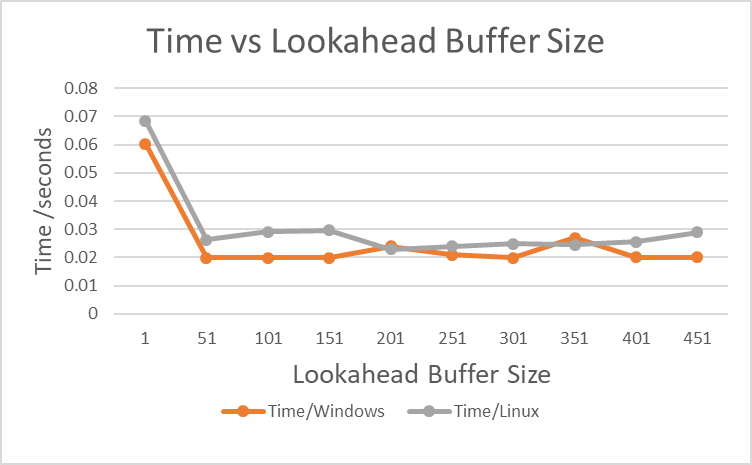
**Results:**

To start of, as the file size of a text file increases the compression time increases as well. This can be seen from the average values of time obtained from the two text files. This is because a larger file will contain more information so the length of the text to be read is larger. For a text file the size of the lookahead buffer plays a more significant role in the compression time. This happens because in my code there is a function which finds the longest match in a string. This function contains a for loop which iterates from the current position in the text up to the size of the lookahead buffer. So by increasing the size of the buffer, the for loop will perform more iterations and thus increase the overall running time. By using a small buffer size, the loop does few iterations and thus producing faster results. Furthermore the window size also plays a role in the compression time of the algorithm, however the buffer size has a grater rate of change than the window size. As the window size increases the running time increases as well. This is because as the window size increases the algorithm has to look further back in the text to find any matches and thus needing more time to read the whole text and compress it. In addition the algorithm runs better on Windows instead of linux. One reason is that I used the university’s Linux timeshare service (Mira) which at that time it might had a lot of users connected to it.

Moreover, the algorithm can also compress images. However images with smaller size than a text file take longer to compress. The reason is that each point in an image consists of RGB values. So, the algorithm compresses three values per point instead of one value per character as the text file. The same procedure occurs with image files as with text files. The lookahead buffer has a higher rate of change than the window size and as the buffer increases the compression time increases as well.

Lastly when having large files, the index data structures used to perform compression grow linearly as the input size increases and thus on big files they can start to exceed the size of memory available.  
**Conclusion:**  
These results show that when using a small lookahead buffer size the compression can be done very fast however this does not mean that the file is compressed well. There is a trade of, between compression speed with the compression ratio. If the look ahead buffer size is small, then the longest match will not be found resulting in more tuples and more memory will be needed to encode the tuples, so the compression ratio becomes worse. However, time wise the algorithm will run very fast since it does only a few iterations. Also by having a smaller window the algorithm will not look far back in the text and thus compress the file faster. Generally, the ideal sizes are to use a relatively big window size usually around a few thousands and a small lookahead buffer usually around one hundred symbols.

1. ***Running time of the decoder (decompression)***

*******File name:* **22.txt** *File size:* ***38.1KB***

**Max** time Windows: 0.109 sec

**Min** time Windows: 0.020 sec

**Average:** 0.033 sec

**Max** time Linux: 0.076 sec

**Min** time Linux: 0.022 sec

**Average:** 0.034 sec

**Max** time Windows: 0.060 sec

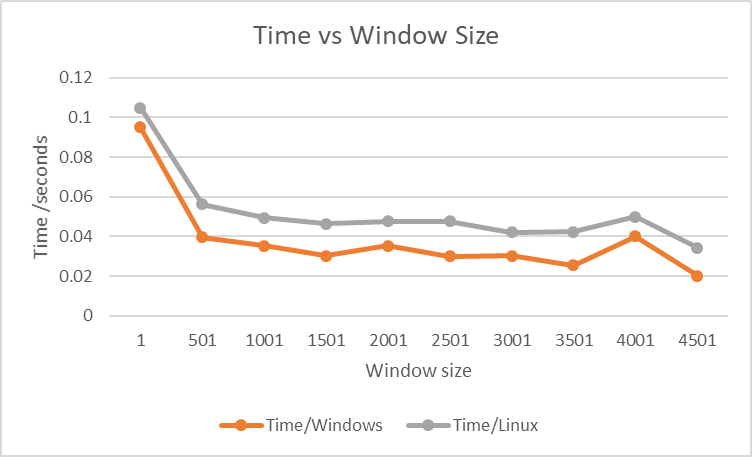
**Min** time Windows: 0.020 sec

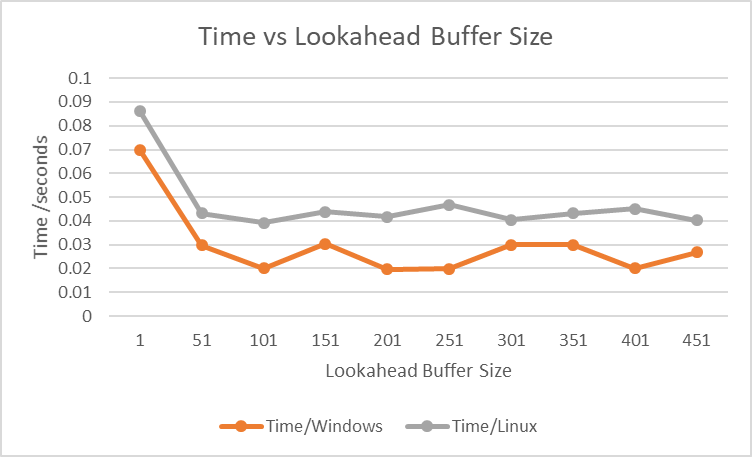
**Average:** 0.025 sec

**Max** time Linux: 0.069 sec

**Min** time Linux: 0.023 sec

**Average:** 0.030 sec

*File name:* **memetics.txt** *File size:* ***46.5KB***



**Max** time Windows: 0.095 sec

**Min** time Windows: 0.020 sec

**Average:** 0.038 sec

**Max** time Linux: 0.105 sec

**Min** time Linux: 0.034 sec

**Average:** 0.052 sec

**Max** time Windows: 0.070 sec

**Min** time Windows: 0.020 sec

**Average:** 0.030 sec

**Max** time Linux: 0.086 sec

**Min** time Linux: 0.039 sec

**Average:** 0.047 sec

*File name:* *File size:*

The experiments that I carried out were done by having a constant window size and changed the buffer size at constant intervals and then vice versa.

**Results:**

The decoding process is simpler than the encoding process since there is no comparisons involved in the decoding. As we can see from the results as the file size of the text files increases the decompression time increases as well. This is the case since a larger file will contain more information and thus more tuples will be used to compress the file. As a result of the increased number of tuples the time needed to decompress the file is greater since the algorithm will have to read more tuples and output the text. Also, a very large file was not used during testing since during decompression when copying characters from the already decompressed part of the file RAM is used to retrieve those characters. If the decompressed part of the file exceeds the RAM space the computer will use what is called virtual memory from the hard drive and this slows down the decompression to a large amount.

Furthermore, my results show that as the buffer size or the window size increase the time needed to decompress the file is lowered. This is because as the size of either the buffer or the window is increased the bits needed to encode each tuple are increased as well. Since more bits are used for each tuple the total number of tuples that the algorithm has to go through are less and so the loop that iterates through each tuple will do fewer iterations and thus have a better decompression time. My results show that the algorithm performs better on windows rather than Linux. Also the rate of change for both increasing the window size or the buffer size is about the same.

**Conclusion**

The algorithm is asymmetric. This means that compression is slower than decompression since in order to compress, the algorithm has to iterate through the text and find a longest possible match but with decompression the algorithm will read the tuples from a list and output the text.

Based on my experiments of separately increasing the window size and lookahead buffer. As we can from the table as the window size and the lookahead buffer size decreases the time decreases as well. This is because, the number of bits to be used is determined from the window and the buffer size. In order to find how many bits we need to use to encode an iteger we take the log value of the window and the buffer size. The larger the two sizes the more bits we need to encode each number so the compression time increases. The algorithm uses a limited window size to find a match in already read text. If there is a very long string, then many possible matches may be lost. To prevent this the window size can be increased however the compression time will increase as well since the algorithm will go back to the text further to find character matches and thus more time is needed to do that since the function which finds the longest match is responsible for increasing the running time the most.