**Information Theory**

To compress the given file, we used three methods: Prediction by partial matching (PPM), Lempel-Ziv algorithm and Huffman coding algorithm. We wanted to compare between them and check which algorithm has the best compression ratio.

First, we will introduce shortly each used algorithm and the main idea of its implementation. Second, we will compare between all three compression algorithms and determine which one is best for our file (dickens).

**1. Compression Methods**

**Prediction by Partial Matching (PPM)**

PPM is an adaptive statistical data compression technique based on context modeling and prediction. PPM models use a set of previous symbols in the uncompressed symbol stream to predict the next symbol in the stream.

Predictions are usually reduced to symbol rankings. Each symbol (a letter, a bit or any other amount of data) is ranked before it is compressed and, the ranking system determines the corresponding code word (and therefore the compression rate).

Arithmetic encoding takes a sequence of symbols as input and gives a sequence of bits as output. The intent is to produce a short output for the given input. Each input yields a different output, so the process can be reversed, and the output can be decoded to give back the original input.   
In our implementation, a symbol is a non-negative integer. The symbol limit is one plus the highest allowed symbol. For example, a symbol limit of 4 means that the set of allowed symbols is {0, 1, 2, 3}.

The following explains all the submodules of our arithmetic coding implementation:

* The class **ArythmeticCompress** derives a static frequency table and writes it to the decompressed file, and **ArythmeticDecompress** reads the frequency table and uses it to decode all the symbols.
* The classes **AdaptiveArithmeticCompress** and **AdaptiveArithmeticDecompress** start with a flat frequency table and update it after each byte is processed, thus making it reflect the statistics of the file being compressed.
* The classes **PpmCompress** and **PpmDeompress** implement a basic version of PPM. In PPM, the frequency predictions for the next symbol is based on the previous n symbols processed.
* The classed **ArithmeticCoderBase, ArithmeticEncoder** and **ArithmeticDecoder**. implement the basic algorithms for encoding and decoding an arithmetic-coded stream. The frequency table can be changed after encoding or decoding each symbol, as long as the encoder and decoder have the same table at the same position in the symbol stream. At any time, the encoder must not attempt to encode a symbol that has a zero frequency.
* Objects with the interface **FrequencyTable** keep track of the frequency of each symbol and provide cumulative frequencies too. The cumulative frequencies are the essential data that drives arithmetic coding.
* The classes **BitInputStream** and **BitOutputStream** are bit-oriented I/O streams, analogous to the standard bytewise I/O streams. However, since they use an underlying bytewise I/O stream, the bit stream’s total length is always a multiple of 8 bits.

**Huffman Coding:**

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols.

Our implementation of Huffman coding makes a frequency dictionary sorted from low to high value in order to decide which characters need to be merged. It also uses a heap to represent the tree that is built in Huffman coding. The main functions are “compress” that compresses the given text file and outputs its compressed version, and “decompress” that decompresses the compressed text file and outputs a text file that is equal to the original.

**Lempel-Ziv:**

LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a length-distance pair. To spot matches, the encoder must keep track of some amount of the most recent data. The structure in which this data is held is called a sliding window. The encoder needs to keep this data to look for matches, and the decoder needs to keep this data to interpret the matches the encoder refers to.

Our Lempel-Ziv implementation first converts the given text file into its binary form and builds a dictionary of size 256.

**2. Results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Original size | Compressed size | Time of compression | Compression ratio |
| PPM |  |  |  |  |
| Huffman coding |  |  |  |  |
| Lempel-Ziv |  |  |  |  |

As can be seen from the results, PPM model is the best compression algorithm for the given text file (‘dickens.txt’).

**3. The part that unique us in the project**

In the project, we used a compression method that was not taught during the course - the PPM method mentioned above. Through reading articles and researching the subject we found this algorithm that gave us amazing results compared to the other algorithms.

In addition, another thing that sets us apart is the comparison of the different algorithms (Hoffman, LZ and PPM), so we could see which one give us the best compresses for the file we received.