UNIVERSIDADE DE AVEIRO
DEPARTAMENTO DE ELECTRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

Information and Coding (2024/25)

Lab work nº 2 — Due: 16 Dec. 2024

Intro

- All programs should be implemented in C++. The comprehensive C++ reference is available at https://en.cppreference.com
- Use a github (or equivalent) repository to manage the development of your software.
- The main goal of this project is to implement efficient audio, image and video encoders.

Part I – BitStream class

The goal of this task is to develop a reusable and efficient C++ class, BitStream, which will allow for precise manipulation of bits when reading from and writing to files. This BitStream class will serve as a foundational tool in later tasks where efficient bit-level operations are essential, especially in implementing codecs.

T1 - Implementing the BitStream class

You will create a C++ class called BitStream, designed to handle bit-level file operations. This class will be used extensively in subsequent tasks, so it must be optimized for speed and memory efficiency. The BitStream class should enable reading and writing of bits in a file while maintaining the bit order from most significant to least significant within each byte.

The class should implement the following core methods:

1. `writeBit` - Writes a single bit to the file.
2. `readBit` - Reads a single bit from the file.
3. `writeBits` - Writes an integer value represented by N bits to the file, where $0 < N < 64$.
4. `readBits` - Reads an integer value represented by N bits from the file, where $0 < N < 64$.
5. `writeString` - Writes a string of characters to the file as a series of bits.
6. `readString` - Reads a string of characters from the file as a series of bits.

You may add additional methods if necessary, but keep in mind the efficiency of this class, as it will be central to the functionality of future codecs.

Specifications:

- The BitStream class should pack bits into bytes efficiently when writing to a file. When reading, it should extract bits in the correct order from most significant to least significant within each byte.
- Error handling: Ensure appropriate error handling for edge cases, such as reading beyond the end of the file.
- Efficiency: Aim to optimize the class for efficient bitwise operations, as this class will be used extensively in later tasks where performance is critical.

T2 - Testing the BitStream class

The goal of this task is to validate the functionality and performance of the BitStream class developed in Task T1. You will implement a simple encoder and decoder program to convert a text file of binary digits (0s and 1s) into a true binary file, and vice versa. This task serves as a practical test of the BitStream class and will help verify its correctness and efficiency in bit manipulation operations.

Using the BitStream class, implement two programs, encoder and decoder:

1. Encoder: The encoder should take a text file containing only the characters '0' and '1' and convert this file into a binary file. Each group of eight characters (bits) in the text file should be packed into one byte in the output binary file. The output binary file will thus contain an efficient, compact binary representation of the text file.
2. Decoder: The decoder should take the binary file produced by the encoder and reconstruct the original text file of 0s and 1s. Each byte in the binary file should be converted back into a sequence of eight characters in the text file.

Specifications:

- The encoder should read the text file bit-by-bit, using the BitStream class to write the bits to a binary file.
- The decoder should read the binary file using the BitStream class, extracting bits to recreate the original sequence of 0s and 1s in a text file.
- Padding: If the number of bits in the text file is not a multiple of 8, add 0s to complete the final byte in the binary file. The decoder should ignore any padded bits when reconstructing the original text file.

Part II – Golomb Coding

In this part, you will develop a C++ class for Golomb coding, a type of universal code often used in data compression. Golomb coding is particularly effective for encoding integers with a geometric distribution and can handle both positive and negative values. You will implement a Golomb encoder and decoder using the BitStream class from Part I for bit-level operations. Afterward, you will test the Golomb encoder using a sample text file containing both positive and negative integers.

T1 - Implementing the Golomb coding class

Develop a C++ class for Golomb coding. This class should include functions for encoding and decoding integers, utilizing the BitStream class for efficient bit-level manipulation. The Golomb coding scheme requires a parameter `m`, which controls the encoding length, making it adaptable to specific probability distributions. Your implementation should handle both positive and negative integers. For negative values, provide two encoding approaches (configurable by the user):

1. Sign and magnitude: encode the sign separately from the magnitude.
2. Positive/negative interleaving: Use a zigzag or odd-even mapping to interleave positive and negative values, so all numbers map to non-negative integers.

Specifications:

- Bit efficiency: use the BitStream class for efficient bit manipulation.
- Parameterization: ensure the class can easily switch between modes and values of `m` for flexibility in encoding different distributions.
- Handling negative values: Implement both sign and magnitude and interleaving for representing negative numbers.

T2 - Testing the Golomb Encoder and Decoder

To verify the functionality of the Golomb encoder, you will create a program that reads a list of integers from a text file, encodes each integer using the Golomb class, and then decodes it to confirm that the original values are accurately reconstructed. This test program will use the following steps:

1. Read: open a text file containing integers, both positive and negative, one per line.
2. Encode: use the Golomb encoder to encode each integer, writing the encoded bits to a binary file via the BitStream class.
3. Decode: read the binary file, decode each integer, and verify it matches the original integer in the input file.
4. Output: print a summary of the encoding and decoding results, highlighting any mismatches (if any) and confirming the accuracy of the implementation.

Part III – Audio Encoding with Predictive Coding

In this part, you will implement an audio encoder that uses predictive coding in combination with Golomb encoding to compress audio files. You will develop both lossless and lossy compression methods.

T1 - Implementing a lossless audio codec using predictive coding

Develop a lossless audio codec that encodes the prediction residuals of audio samples using Golomb coding. This codec should handle both mono and stereo audio files.

For mono audio files use temporal prediction: Predict each sample based on one or more previous samples in the same channel.

For stereo audio files use both temporal and inter-channel prediction: For each channel, predict each sample based on previous samples within the same channel and optionally use information from the other channel.

The efficiency of Golomb coding relies on an optimal choice of the parameter `m`. You may implement a fixed `m` parameter that is specified by the user, or, preferably, develop an adaptive mechanism to dynamically determine the optimal `m` value during encoding and decoding.

Specifications:

- BitStream integration: use the BitStream class to handle bit-level reading and writing of encoded data.
- Parameter `m`: support both fixed and adaptive selection of the parameter `m`.
- Audio Format: the codec should handle common PCM audio formats, such as .wav files.

T2 - Implementing lossy audio coding with bitrate control

Extend the audio codec from Task T1 to include an option for lossy encoding. The lossy codec should be able to adjust the compression level based on a target average bitrate specified by the user.

The lossy encoder should use quantization to achieve the desired bitrate. Quantization reduces the precision of each sample, which reduces the amount of data required to encode it. The encoder should aim to meet the target bitrate as closely as possible, adjusting quantization levels dynamically if necessary.

Specifications:

- Bitrate control: allow the user to specify an average bitrate for the encoded audio.
- Quantization: implement quantization to reduce precision and meet the specified bitrate.
- BitStream and Golomb integration: use the BitStream and Golomb classes to store quantized data efficiently.
- Adaptability: implement a mechanism to adjust quantization levels if the target bitrate is not being met.

Further extensions - Implementing Lossy Audio Codec with Transform Coding (DCT)

In this task, you will implement a lossy audio codec for mono audio files using Discrete Cosine Transform (DCT) based on block-by-block processing. The goal is to achieve good compression while preserving audio quality as much as possible.

Each audio block will be transformed into the frequency domain using the DCT, and the resulting coefficients will be quantized. After quantization, the coefficients should be encoded using the BitStream class and written to a binary file. The decoder will reconstruct the audio by reading from the binary file, applying inverse quantization, and performing the Inverse DCT (IDCT) to obtain an approximation of the original audio.

Specifications:

- Block size: choose an appropriate block size (e.g., 256 or 512 samples) for DCT processing.
- Quantization: quantize DCT coefficients to reduce the number of bits, achieving compression.
- BitStream integration: use the BitStream class to store quantized DCT coefficients efficiently.
- Decoding requirement: the decoder should rely only on the binary file produced by the encoder to reconstruct the audio.

Part IV – Image and Video Coding with Predictive Coding

In this part, you will implement an image and video encoder applying predictive coding techniques for both lossless and lossy encoding.

T1 - Lossless image coding using predictive coding

Implement a lossless image codec for color images using Golomb coding on the prediction residuals. Select appropriate predictors to minimize prediction error, resulting in more efficient Golomb encoding. Your codec should:

- Use spatial prediction to estimate pixel values based on neighboring pixels (ex. JPEG and JPEG-LS predictors).
- Encode the residuals (differences between predicted and actual pixel values) using Golomb coding with optimized values of `m`.

T2 - Intra-Frame Video Coding

Adapt the lossless image codec from Task T1 to support intra-frame video coding, encoding videos as sequences of images. Each frame should be encoded independently (intra-frame coding), using only spatial prediction within each frame.

Ensure that the encoded video file includes all necessary parameters (such as image dimensions and Golomb parameter m) for decoding.

T3 - Inter-Frame Video Coding

Expand the codec from Task T2 to support inter-frame (temporal) prediction with motion compensation. The codec should use intra-frames (I-frames) at regular intervals and inter-frames (P-frames) that encode only changes relative to the previous frame.

Requirements:

1. Intra-frame Periodicity: Let the user specify the interval for I-frames.
2. Block Size and Search Area: Let the user specify the block size and search range for motion estimation.
3. Mode Decision: For each block in P-frames, decide whether to use intra or inter mode based on the resulting bitrate, encoding blocks in intra mode if they result in a lower bitrate.

T4 - Lossy Video Coding with Quantization of Prediction Residuals

Extend the codec to support lossy video coding by introducing quantization on the prediction residuals before applying Golomb coding. Quantization will reduce the precision of residuals, achieving better compression at the cost of some data loss.

Requirements:

- Allow the user to specify the quantization level, which controls the trade-off between compression and quality.
- Quantize the prediction residuals before encoding them with Golomb coding.

Further extensions - DCT-Based Transform Coding

Implement a DCT-based transform coding for lossy video compression. In this task, you will apply the Discrete Cosine Transform (DCT) to each block of a frame, quantize the DCT coefficients, and then encode the coefficients using zig-zag ordering to prioritize low-frequency components.

Steps:

1. DCT transformation: divide each frame into blocks (e.g., 8x8 pixels) and apply the DCT to each block, transforming the spatial data into frequency components.
2. Coefficient quantization: quantize the DCT coefficients to reduce data, preserving more significant low-frequency coefficients while discarding or heavily quantizing high-frequency components.
3. Zig-Zag ordering: use zig-zag ordering on the quantized DCT coefficients, placing the low-frequency coefficients at the start, which tend to carry more important information.
4. BitStream encoding: write the quantized coefficients to a binary file using the BitStream class.

The decoder should rely only on the binary file to reconstruct an approximate version of the original video, reversing the process with inverse quantization and Inverse DCT (IDCT).

Part IV – Final Report and Presentation

The objective of this part is to compile a comprehensive report and prepare a final presentation that showcases the results and effectiveness of the various codecs implemented in the project. This is an opportunity to critically evaluate the performance of your work, compare it with industry standards, and demonstrate the functionality of your software.

T1 – Report

Create a report that documents all the significant steps, design decisions, and findings from each task in the project. This report should focus on the outcomes and analysis of your implementation, rather than a detailed description of the code itself.

Key elements to include:

1 - Summary of each task: for each part of the project (image, audio, and video coding), briefly summarize the goals, key decisions, and approaches taken.

2 - Performance metrics:

- Processing time: measure the encoding and decoding times for various files, discussing any optimizations or bottlenecks encountered.
- Compression ratios: include compression ratios achieved for each codec, and how they vary depending on the type of data (e.g., text, image, audio, video).
- Error metrics: for lossy codecs, calculate the error introduced during compression/decompression (e.g., Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR) for images and audio).

3 - Comparative Analysis:

- Compare the results of your codecs with existing industry-standard codecs (e.g., JPEG for images, MP3 for audio, H.264 for video).
- Discuss any differences in compression efficiency, quality, and processing time, and analyze why those differences occur.

4 - Limitations and improvements: Reflect on the limitations of each codec implementation and suggest possible improvements.

Specifications:

- Structure: organize the report by sections that correspond to each part of the project, with dedicated sections for performance metrics, analysis, and comparison.
- Clarity: ensure that technical terms and metrics are well-defined, and include visual aids (graphs, tables) where appropriate.
- Brevity: keep descriptions concise and focused on outcomes and analysis rather than code details.

T2 – Final presentation and demonstration

Prepare a presentation and demonstration for the final exam day in January, showcasing the functionality and results of your project.

1 - Presentation outline:

- Introduction: brief overview of the project objectives and methods.
- Highlights of each codec: summarize the approach and key results for each codec (image, audio, video).
- Performance and comparisons: present the key findings on processing time, compression ratio, and error metrics, comparing your results with industry-standard codecs.
- Challenges and solutions: discuss any major challenges faced and the solutions you implemented.
- Future improvements: conclude with ideas for future improvements based on the project outcomes.

2 - Demonstration:

- Demonstrate the working of at least one codec from each category (image, audio, and video).
- Show real-time encoding and decoding of sample files, highlighting the achieved compression and, for lossy codecs, the quality degradation.

Specifications:

- Demonstration setup: ensure your software is fully operational and have sample files prepared in advance for the demo.
- Time management: plan to complete the presentation and demonstration within the allocated time (typically around 15-20 minutes).

Alternatives to the further extensions

Usually, the purpose of studying data compression algorithms is twofold. The need for efficient storage and transmission of data is often the main motivation. However, underlying every compression technique, there is a model that tries to reproduce as closely as possible the information source to be compressed. This model may be interesting on its own, as it can shed light on the statistical properties (or, more generally, algorithmic properties) of the source.

One of the most used approaches for representing data dependencies relies on the use of Markov models. In lossless data compression, we use a specific type called a discrete-time Markov chain or finite-context model. A finite-context model can be used to collect high-order statistical information of an information source, assigning a probability estimate to the symbols of the alphabet according to a conditioning context computed over a finite and fixed number of past outcomes. Our main goal is to predict the next outcome of the source. To do this, we infer a model based on the observed past of the sequence of outcomes.

Consider now the problem of determining the "similarity" between a target text, t, and some reference texts, r_i. For example, each r_i could be a sample text from a certain language, and t could be a text whose language needs to be determined. Usually, the traditional approach to solving this classification problem begins with feature extraction and selection operations. The features obtained are then fed into a function that maps the feature space onto the set of classes and performs the classification. One of the most difficult parts of this problem is choosing the smallest set of features that retains enough discriminant power to tackle the problem.

The representation of the original data by a small set of features can be seen as a special form of lossy data compression. This suggests a question: Can data compression be explicitly used to approach classification problems, removing the need for a separate feature extraction stage? The answer is affirmative; it is possible to adopt an information-theoretic approach to the classification problem, bypassing the feature extraction and selection stages. In other words, compression algorithms can be used to measure similarity between files.

The idea is the following: For each class, represented by the reference text r_i, we create a model that is a good description of r_i. By a "good description," we mean a model that requires fewer bits to describe r_i than other models, or, in other words, that is a good compression model for the "members of the class" r_i. Then, we assign to t the class corresponding to the model that requires the fewest bits to describe it, i.e., to compress t."

Part I

Develop a program, named fcm, with the aim of collecting statistical information about texts, using finite-context models. The order of the model, `k`, as well as the smoothing parameter, α, should be parameters passed to the program. This program should provide the entropy of the text, as estimated by the model.

Part II

1 - Develop a program, named lang, that accepts two files: one, with a text representing the class r_i (for example, representing a certain language); the other, with the text under analysis, `t`. Modeling should be performed using the finite-context models implemented in Part I. Other parameters, such as the order of the context model and the parameter α of the probability estimator, should also be provided to the program. The program should report the estimated number of bits required to compress `t`, using the model computed from r_i.

2 - Based on the lang program, build a language recognition system, findlang, that, from a set of examples from several languages (the r_i), provides a guess for the language in which a text `t` was written. You should obtain results with as many different languages as possible (definitely, more than ten). It is up to you to find texts that are good representatives of the language and test them with appropriate examples.

3 - As a bonus challenge, explore the possibility of using combinations of several finite-context models (with different orders) to represent each language.