UNIVERSIDADE DE AVEIRO

DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

42078 - Informação e Codificação

Eduardo Fernandes 98512

João Ferreira 103625

Pedro Monteiro 97484

Lab Work nº3

2023

Contents

List of Code Blocks

1 Exercise 1

1.1 Exercise Description

In this exercise, we will explain the process, application, and verification of our lossless image codec.

1.2 Encoder and Decoder

Our encoder has 8 compression modes, and after various tests, we found that mode 8 provides the best compression in a majority of cases.

For each pixel in the image, this mode is calculated three times because each pixel is assigned three values. The compression modes were as follows:

- Mode 1 $\rightarrow X = a$
- Mode 2 $\rightarrow X = b$
- Mode 3 $\rightarrow X = c$
- Mode 4 $\rightarrow X = a + b - c$
- Mode 5 $\rightarrow X = \frac{a(b-c)}{2}$
- Mode 6 $\rightarrow X = b + \frac{(a-c)}{2}$
- Mode 7 $\rightarrow X = \frac{(a+b)}{2}$

For improved compression, a dynamic M was assigned to the algorithm function of the codec, based on the Golomb class. For this, we have an input parameter in our encoder where we have an X and a Y. Its effect is that every X iterations of the already calculated values, a new M value is calculated based on the last Y values.

Note: X should be greater than or equal to Y.

After several tests using scripts, we were able to assign a better default value for X and Y, and this value is 2500 for both. The default encoding mode is mode 8. All this encoded value is stored in a string and then saved in a binary file through a write_bin_to_file function. The decoding process is exactly the reverse of all the mentioned processes above.

1.3 Usage

To execute this program, provide the path of the ppm file you want to encode and the corresponding file to save these data. On the other hand, for decoding, it is necessary to provide the file where the data were saved and the name of a ppm file that will serve as the output.

```
$./3encoder_image <input.ppm> <output>
$./3decoder_image <output> <output.ppm>
```

1.4 Results

As a lossless codec, there can be no losses in the values assigned to pixels. To ensure this, the hash of the original file was calculated and compared to the hash of the decoded file, verifying a 0% loss.

The compression ratio was calculated using the following formula:

$$\text{Compression} = \frac{\text{EncodedFile}}{\text{OriginalFile}}$$

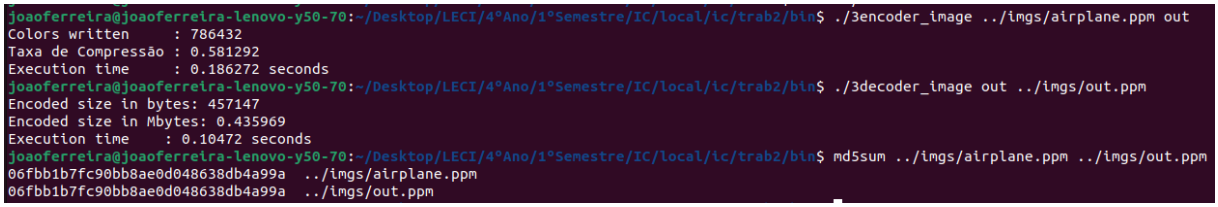

Figure 1: Encoding and Decoding features

The mode values were tested on various images. The value decreases from mode 1 to mode 8, with mode 8 obtaining the best results.

2 Exercise 2

2.1 Exercise Description

This section is dedicated to the implementation of an intraframe codec for the YUV4MPEG2 format (.y4m), which can be compressed with the following compression formats: YUV420 and YUV422. This codec has options to choose from 8 modes of inter-pixel value prediction and the periodicity of updating the value of m.

2.2 Supported YUV Formats

2.2.1 YUV420

In this compression format, in the YCbCr color space, each 2x2 block of pixels in an image is represented by 4 Y samples, one for each pixel, but all 4 pixels share the same Cb and Cr samples. In other words, to represent 4 pixels, 6 values are required.

2.2.2 Encoding and Decoding

For the video encoding process, a header is first created containing all the necessary information for the decoding process. Subsequently, the encoding of each frame of the video takes place. Thus, the encoding process for an individual frame involves encoding the values of the Y space. Starting with the first column and first row, the original values belonging to that region are encoded. For the remaining values in that color space, the residue of the Y value of the respective pixel is calculated, and the result is then encoded.

In turn, as blocks of 2x2 pixels share the same Cb and Cr values, the frame is divided into blocks of this dimension. Each of these blocks is treated as if it were a single pixel during the encoding process. In the encoding of this 'pixel,' the Cb value is first encoded, followed by the Cr value. Similar to the encoding of Y space values, the values of pixels in the first column (Cb and Cr) are initially encoded, followed by the values of pixels in the first row (Cb and Cr). Finally, the residual calculation of the values (Cb and Cr) for the remaining pixels is performed, and the resulting values are encoded.

Upon completion of the encoding process, a file is generated containing the header and the encoded information.

For the decoding process, the header of the encoded file is first read, containing all the necessary information to decode the file. Subsequently, the frame-by-frame decoding process is initiated. In this frame decoding process, the original values of the first column and first row of the Y color space are initially read. Through the inverse operations of the residual calculation for each pixel, the original value of Y for that pixel is obtained.

For the Cb and Cr color spaces, the process is similar to that of the Y color space, with the difference that the decoding process for these two channels is done simultaneously. This is due to the fact that when reading a value from the Cb channel, the next value belongs to the Cr channel, and each pair of Cb and Cr values corresponds to a 2x2 pixel block.

Upon completing this decoding process, the resulting frames are used to recreate the previously encoded video.

2.3 Usage

To run this program, provide the path of the y4m file you want to encode and the corresponding txt file to save these data. On the other hand, for decoding, it is necessary to provide the file where the data were saved and the name of a y4m file that will serve as the output.

```
$./codec_video_tests encode YUV420 <file.y4m> <out.txt>

$./codec_video_tests decode  <out.txt> <decoded.y4m>
```

2.4 Results

2.4.1 Compress Ratio

The Compression Ratio is a simple numerical representation of the ”compression power” of specific codecs or compression techniques. It is a measure of the relative reduction in the size of the data representation produced by a data compression algorithm. Typically, it is expressed as the ratio of the uncompressed size to the compressed size.

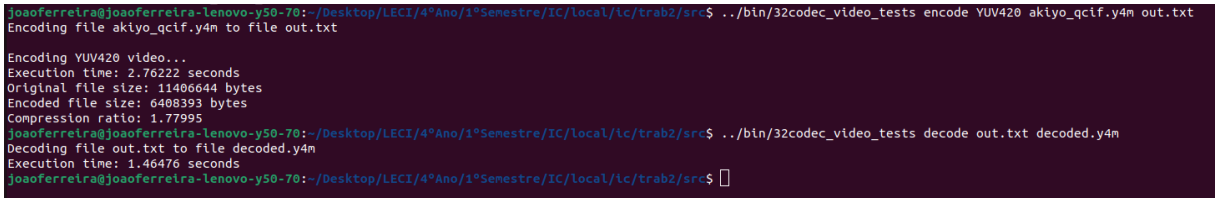

Figure 2: YUV420 Encoding and Decoding features

If a video has a lot of motion and/or a distinct color palette, it will have a lower compression ratio compared to another video with little motion and few colors.

2.4.2 Prediction Residuals

As mentioned earlier, this codec has 8 different ways to calculate the predicted value of a pixel, which, once calculated, will be used to obtain the residue value.

Analyzing the compression ratio, we can observe that depending on the mode used, the compression ratio increases for higher-numbered modes and is smaller for lower modes. This is due to the complexity of the calculations performed to determine the predicted value. The choice of the method for calculating the predicted value is also related to the encoding and decoding time, as more complex calculations require more time to be performed compared to simpler calculations.

3 Exercise 3

3.1 Exercise Description

This exercise involves developing a video codec that uses Golomb coding for encoding grayscale video sequences, expanding the codec to support inter-frame prediction, which is more widely recognized as motion compensation. In addition to this, the encoder's parameters require specific attention:

- Firstly, the intra-frame periodicity, concerning I or key frames, should be modifiable and controllable through an input parameter.
- Secondly, users must have the ability to define the block size and the search area for coding inter-frames, also known as P frames.
- Finally, a crucial aspect for the encoding of P frames is the codec's capability to decide whether to encode a current block in intra or inter mode. This decision should be based on the bitrate that is being produced.

3.2 Code

3.2.1 video_utils.h

This header file,*video_utils.cpp*, defines the codec_video class, which is intended for encoding and decoding video streams.

The struct *MotionVector* is defined to represent motion vectors, which are crucial in video compression techniques like motion estimation. It simplifies the representation and handling of these vectors.

Code Block 1: Motion Vector Struct

```
// structure to represent motion vectors with horizontal (dx) and
   vertical (dy) components
struct MotionVector {
    int dx, dy;
};
```

The public interface of the codec_video class offers a range of functionalities for initializing the codec with different configurations, encoding and decoding video streams, and handling specific video formats (YUV420, YUV422).

Code Block 2: Codec Video Class Public Section

```
public:
    // constructors
    codec_video(int mode, uint32_t p, int quantization, int blockSize
       , int searchArea);
    codec_video(int mode, uint32_t p, int quantization);
    codec_video();
    codec_video(int mode, uint32_t p);

    // method to encode a video file
    int encode_video(const char *fileIn, const char *fileOut, const
       string &yuv_format);
```

```
10
11      // method to decode a video file
12      int decode_video(const char *fileIn, const char *fileOut);
13
14      // static methods for writing YUV420 and YUV422 formatted frames
           to a file
15      static int YUV420_write(const char *fileOut, uint32_t n_frames,
           uint32_t height, uint32_t width, Mat *to_write, string header)
           ;
16      static int YUV422_write(const char *fileOut, uint32_t n_frames,
           uint32_t height, uint32_t width, Mat *to_write, string header)
           ;
17
18      // static methods to read and write binary data to and from a
           file
19      static string read_bin_from_file(const char *fileIn);
20      static int write_bin_to_file(const char *fileOut, const string &
           encoded);
```

The private section of codec_video is essential for maintaining the integrity and functionality of the codec. It includes state variables that configure and define the codec's behavior, along with a suite of methods for core processing tasks such as motion estimation, residual calculation, and frame encoding.

Code Block 3: Codec Video Class Private Section

```
1  private:
2      // private members to store codec settings and state
3      int mode;
4      uint32_t period;
5      int quantization;
6      int blockSize;
7      int searchArea;
8      MotionVector prevMotionVec;
9
10     // private methods for internal processing
11     int calculateSAD(const Mat &block1, const Mat &block2);  //
          calculates the Sum of Absolute Differences
12     MotionVector motion_estimation(Mat currentBlock, Mat refFrame,
          int searchArea);  // estimates motion vector for a block
13     bool should_encode_intra(Mat currentBlock, Mat refFrame, int
          threshold);  // determines whether to use intra-frame encoding
14     string encode_motion_vector(MotionVector mv, golomb &encoder);
          // encodes a motion vector
15     string encode_residual(const Mat &residual, golomb &encoder);  //
           encodes residual data
16     Mat compute_residual(const Mat &currentBlock, const Mat &refBlock
          , const MotionVector &mv);  // computes residual block
17     string encodePFrame(const Mat &currentFrame, const Mat &refFrame,
           golomb &encoder);  // encodes P-frames
18     string encodeIFrame(const Mat &frame, golomb &encoder);  //
          encodes I-frames
19     void intToByteArray(int value, char *byteArray);  // converts an
```

```
        integer to a byte array
20
21      // methods for decoding
22      MotionVector decode_motion_vector(string bitstream, golomb &
            decoder);  // decodes a motion vector from a bitstream
23      Mat decode_residual(string bitstream, int rows, int cols, golomb
            &decoder);  // decodes residual data from a bitstream
24      double calculateBlockVariance(const cv::Mat &block);  //
            calculates the variance of a block
```

3.2.2 codec_video.cpp

This is the implementation file for the codec_video class. This implementation file provides a comprehensive set of functionalities for video encoding and decoding, including detailed control over encoding parameters, motion estimation, handling different frame types (I-frames and P-frames), and dealing with specific video formats (YUV420 and YUV422). It leverages the capabilities of OpenCV for video frame manipulation and Golomb coding for efficient data compression.

The functions outlined in the header file are implemented here, along with additional helper functions that are utilized throughout the program. The constructors for the class are also implemented in this section.

Code Block 4: Constructor Example

```
1  codec_video::codec_video(int mode, uint32_t p, int quantization, int
       blockSize, int searchArea)
2  {
3      this->mode = mode;
4      this->period = p;                    // frame period for certain
           processing
5      this->quantization = quantization; // quantization parameter for
           compression
6      this->blockSize = blockSize;         // size of the block for
           processing
7      this->searchArea = searchArea;       // area to search for motion
           estimation
8      prevMotionVec = {0, 0};              // initialize previous motion
           vector to zero
9  }
```

3.2.3 main_intra_encoder.cpp

This is the main application for the video encoding program. It uses the codec_video class, defined in the ”video_utils.h” header file, to encode video files.

The code begins by checking the correct arguments are passed to the program. Then parses the command-line arguments to extract the necessary parameters for the video encoding process. These include the paths of the input and output video files, the YUV format (a standard format in video processing), and various encoding parameters like mode, period, quantization level, block size, and search area.

Code Block 5: Main Encoder Initial Code

```
if (argc != 9) // check if the correct number of arguments is passed
{
    cerr << "Usage: " << argv[0] << " <input_file> <output_file> <
       yuv_format> <mode> <period> <quantization> <block_size> <
       search_area>" << endl;
    return -1;
}

// parse command line arguments
string input_file = argv[1];
string output_file = argv[2];
string yuv_format = argv[3];
int mode = stoi(argv[4]);
int period = stoi(argv[5]);
int quantization = stoi(argv[6]);
int block_size = stoi(argv[7]);
int searchArea = stoi(argv[8]);
```

A codec_video object, named videoCodec, is created using the parsed parameters. This object is responsible for handling the video encoding process according to the specified settings.

Code Block 6: Codec Video Object

```
// create a codec_video object with the provided parameters
codec_video videoCodec(mode, period, quantization, block_size,
   searchArea);

cout << "Encoding video..." << endl;
if (videoCodec.encode_video(input_file.c_str(), output_file.c_str(),
   yuv_format) != 0)
{
    cerr << "Error during video encoding." << endl;
    return -1;
}
```

It then calls the encode_video method on the videoCodec object, passing the input and output file paths and the YUV format. The *codec_video::encode_video* function is a comprehensive method within the codec_video class, designed for encoding video files. It takes three parameters: paths to the input and output files (fileIn and fileOut, respectively) and the YUV format (yuv_format).

It attempts to open the input video file using OpenCV's *VideoCapture* class. Upon successful opening of the video file, the function retrieves essential properties of the input video, such as its frame width, height, and the total number of frames.

Code Block 7: Open Input File

```
VideoCapture cap(fileIn);
if (!cap.isOpened())
{
    cerr << "Error opening video file" << endl;
    return -1;
}
```

```

// get properties of the input video
Size frameSize = Size((int)cap.get(CAP_PROP_FRAME_WIDTH), (int)cap.
    get(CAP_PROP_FRAME_HEIGHT));
uint32_t frameCount = (uint32_t)cap.get(CAP_PROP_FRAME_COUNT);
```

A Golomb encoder is then initialized with a predefined M-value, an important step for efficient data compression. The output file is opened in binary mode for writing the encoded video data.

Code Block 8: Initialize Golomb Encoder

```
// initialize Golomb encoder with a predefined M-value
golomb golombEncoder(1000);

ofstream outputFile(fileOut, ios::binary);
if (!outputFile.is_open())
{
    cerr << "Error opening output file" << endl;
    return -1;
}
```

The next step involves writing metadata to the output file. This metadata includes the video's dimensions, frame count, YUV format code, and various encoding settings like mode, quantization level, block size, and search area. This information is crucial to decode the video.

Code Block 9: Write Settings

```
// convert and write video properties to the output file
char buffer[4];
intToByteArray(frameSize.width, buffer);
outputFile.write(buffer, 4);
intToByteArray(frameSize.height, buffer);
outputFile.write(buffer, 4);
intToByteArray(frameCount, buffer);
outputFile.write(buffer, 4);

int yuvFormatCode = (yuv_format == "YUV420") ? 1 : (yuv_format == "
    YUV422") ? 2 : 0;
intToByteArray(yuvFormatCode, buffer);
outputFile.write(buffer, 4);

// write encoding settings
intToByteArray(mode, buffer);
outputFile.write(buffer, 4);
intToByteArray(period, buffer);
outputFile.write(buffer, 4);
intToByteArray(quantization, buffer);
outputFile.write(buffer, 4);
intToByteArray(blockSize, buffer);
outputFile.write(buffer, 4);
intToByteArray(searchArea, buffer);
outputFile.write(buffer, 4);
```

The core of the function lies in encoding each frame of the video. It iterates through each frame, deciding whether to encode it as an I-frame or a P-frame based on the frame's position in the sequence and the specified period. The *encodeIFrame* or *encodePFrame* methods are used accordingly to encode each frame. The resulting bitstream for each frame is then written to the output file.

Code Block 10: Encode Frames

```
// iterate over each frame and encode
Mat currentFrame, previousFrame;
for (uint32_t frameIndex = 0; frameIndex < frameCount; frameIndex++)
{
    cap >> currentFrame;
    if (currentFrame.empty())
        break;

    bool isIFrame = (frameIndex % period == 0); // determine if
       current frame is an I-frame or a P-frame
    string frameBitstream = isIFrame ? encodeIFrame(currentFrame,
       golombEncoder) : encodePFrame(currentFrame, previousFrame,
       golombEncoder);
    outputFile.write(frameBitstream.c_str(), frameBitstream.size());
    currentFrame.copyTo(previousFrame);
}
```

The *encodePFrame* and *encodeIFrame* functions are designed for encoding Predictive frames (P-frames) and Intra-coded frames (I-frames), respectively.

The *encodePFrame* function specializes in encoding P-frames. These frames rely on data from previous frames, reducing the amount of information needed for each frame. The function starts by creating an empty string to store the resulting bitstream and sets the block width and height based on the class's blockSize. It then processes the current frame in blocks, adjusting block sizes near the frame edges. For each block, it extracts corresponding blocks from both the current and reference frames. Motion estimation is performed to determine the motion vector, indicating the movement of the current block relative to the reference. This vector is encoded and added to the bitstream. Following this, the function computes and encodes the residual block - the difference between the current block and its predicted version based on the motion vector. The encoded residual is appended to the bitstream, which is then returned, representing the encoded P-frame.

Code Block 11: Encode PFrames

```
for (int y = 0; y < currentFrame.rows; y += blockHeight)
{
    for (int x = 0; x < currentFrame.cols; x += blockWidth)
    {
        // adjust block size for edges
        int actualBlockWidth = std::min(blockWidth, currentFrame.cols
            - x);
        int actualBlockHeight = std::min(blockHeight, currentFrame.
           rows - y);

        Rect blockRect(x, y, actualBlockWidth, actualBlockHeight);

```

```
        // extract the current and reference blocks
        Mat currentBlock = currentFrame(blockRect);
        Mat refBlock = refFrame(blockRect);

        // perform motion estimation
        MotionVector mv = motion_estimation(currentBlock, refBlock,
            this->searchArea);

        // encode the motion vector
        bitstream += encode_motion_vector(mv, encoder);

        // compute and encode the residual block
        Mat residual = compute_residual(currentBlock, refBlock, mv);
        bitstream += encode_residual(residual, encoder);
    }
}
```

The motion_estimation function is designed to estimate the motion vector between a block in the current video frame and a reference frame. The function operates by searching for the position in the reference frame where the block from the current frame matches most closely, based on the Sum of Absolute Differences (SAD) - a measure of similarity between blocks. It iteratively adjusts the search area and refines the process until it finds the position with the minimum SAD, indicating the best match. The resulting motion vector, representing the displacement between the current block and its matching position in the reference frame, is then returned.

Code Block 12: Motion Estimation

```
while (stepSize >= 1) // iteratively search for the block position
    that minimizes the SAD
{
    bool foundBetterMatch = false;

    for (int y = -stepSize; y <= stepSize; y += stepSize)
    {
        for (int x = -stepSize; x <= stepSize; x += stepSize)
        {
            int newX = xCenter + x; // new candidate X position
            int newY = yCenter + y; // new candidate Y position

            // check if the new block is within the frame boundaries
            if (newX < 0 || newY < 0 || newX + blockWidth > refFrame.
                cols || newY + blockHeight > refFrame.rows)
                continue;
            // define a rectangle in the reference frame for block
                matching
            Rect blockRect(newX, newY, blockWidth, blockHeight);
            Mat refBlock = refFrame(blockRect);

            // calculate the SAD for the candidate block
            int sad = calculateSAD(currentBlock, refBlock);

```

```
            // update the best motion vector if a better match is
               found
            if (sad < minSAD)
            {
                minSAD = sad;
                bestMotionVec.dx = newX;
                bestMotionVec.dy = newY;
                foundBetterMatch = true;
            }
        }
    }

    // update the center position to the best match found in this
       iteration
    if (foundBetterMatch)
    {
        xCenter = bestMotionVec.dx;
        yCenter = bestMotionVec.dy;
    }
    else
    {
        // reduce the step size for finer search in the next
           iteration
        stepSize /= 2;
    }
}
```

After all frames have been processed and written to the output file, the file is closed. The function also calculates and displays the total time taken for the encoding process.

Code Block 13: Display Encoding Time

```
outputFile.close();
cout << "Encoding time: " << static_cast<float>(clock() - time_req) /
    CLOCKS_PER_SEC << " seconds" << endl;
```

3.2.4 main_intra_decoder.cpp

The program begins by checking if the correct number of arguments (three) is passed, which includes the program name, the path to the encoded video file, and the path for the output file.

After verifying the arguments, the program parses the command line to extract the paths of the encoded input file and the desired output file for the decoded video. It then creates an instance of the codec_video class.

The program proceeds to decode the video by calling the decode_video method of the codec_video object, passing the paths of the encoded file and the output file.

The decode_video function in the codec_video class is responsible for decoding an encoded video file. The function takes two arguments: paths to the input encoded file and the output file where the decoded video will be saved.

The function proceeds to decode the video metadata from the binary data, extracting crucial information like the video's width, height, number of frames, YUV format code, and

other encoding parameters such as mode, period, quantization level, block size, and search area. This metadata is essential for correctly reconstructing the video frames.

Code Block 14: Decode Function Metadata

```
outputFile.close();
// decode video metadata
uint32_t width = std::stoul(encoded.substr(0, 32), nullptr, 2);
uint32_t height = std::stoul(encoded.substr(32, 32), nullptr, 2);
uint32_t frames = std::stoul(encoded.substr(64, 32), nullptr, 2);
uint32_t yuvFormatCode = std::stoul(encoded.substr(96, 32), nullptr,
    2);
mode = static_cast<int>(std::stoul(encoded.substr(128, 32), nullptr,
    2));
period = static_cast<uint32_t>(std::stoul(encoded.substr(160, 32),
    nullptr, 2));
quantization = static_cast<int>(std::stoul(encoded.substr(192, 32),
    nullptr, 2));
blockSize = static_cast<int>(std::stoul(encoded.substr(224, 32),
    nullptr, 2));
searchArea = static_cast<int>(std::stoul(encoded.substr(256, 32),
    nullptr, 2));
```

Memory is allocated for storing the decoded frames, and the function decodes each frame according to the specified YUV format. For YUV420 or YUV422 formats, it uses specific decoding methods to process each frame and stores the decoded frames in the allocated memory.

Code Block 15: YUV Formats Decode Functions

```
// allocate memory for storing the decoded frames
Mat *to_write = new Mat[frames];
Mat *decoded_frame = new Mat(height, width, CV_8UC3);

if (yuvFormatStr == "YUV420")
{
    // handle YUV420 decoding
    for (uint32_t i = 0; i < frames; i++)
    {
        encoded_ptr = codec_img.YUV420_decode_video_frame(encoded_ptr
            , decoded_frame, height, width);
        to_write[i] = *decoded_frame;
    }
    // write decoded video to file
    if (YUV420_write(fileOut, frames, height, width, to_write, "
        Header") != 0)
    {
        cerr << "Error writing video" << endl;
        return -1;
    }
}
else if (yuvFormatStr == "YUV422")
{
    // handle YUV422 decoding
    for (uint32_t i = 0; i < frames; i++)
```

```
24      {
25          encoded_ptr = codec_img.YUV422_decode_video_frame(encoded_ptr
                , decoded_frame, height, width);
26          to_write[i] = *decoded_frame;
27      }
28      // write decoded video to file
29      if (YUV422_write(fileOut, frames, height, width, to_write, "
            Header") != 0)
30      {
31          cerr << "Error writing video" << endl;
32          return -1;
33      }
34  }
35  else
36  {
37      cerr << "Error: YUV format not supported: " << yuvFormatStr <<
            endl;
38      return -1;
39  }
```

After decoding all frames, the function attempts to write the decoded video to the output file.

3.3 Usage

To run the encode program are necessary eight command line arguments: path to the input video file, path for the output video file, YUV format of the video, mode of the codec, frame period for processing, quantization parameter for compression, size of the block for processing, and the area to search for motion estimation:

```
$ ./video_encoder <input_file> <output_file> <yuv_format> <mode> <
    period> <quantization> <block_size> <search_area>
```

On the other hand, the decoder needs only two arguments, the path to the binary file and the name of the output video file:

```
$ ./video_decoder <encoded_file> <output_file>
```

3.4 Results

The table 1 shows the results from a series of video encoding experiments with varying periods, block sizes, and search areas. The period shows a clear pattern where a lower period results in a larger output file size, Figure 3. This makes sense since I-frames are encoded without reference to other frames and are typically larger in size.

Regarding the search area, a smaller value correlates with a larger output size, which could imply that a more extensive search area could lead to more effective motion estimation and thus better compression. However, the effect of the search area on the output size isn't very pronounced. As expected, larger block sizes and search areas increase the encoding time due to the higher computational load required for motion estimation.

When it comes to reducing the output size, increasing the period seems to be the most effective strategy given this dataset. It significantly reduces the output size, indicating fewer

I-frames are being used. On the other hand, the search area's impact on the output size is less consistent, but a larger search area seems to reduce the output size slightly

Period	Block Size	Search Area	Time (ms)	Output size (MB)
garden.sif.y4m				
3	16	4	50001	96.03
3	16	8	4958	96.03
3	16	16	5205	96.03
3	32	4	4938	95.69
3	32	8	5042	95.69
3	32	16	5001	95.69
5	16	4	5065	94.94
5	16	8	5044	94.94
5	16	16	5296	94.94
5	32	4	5040	94.52
5	32	8	5019	94.52
5	32	16	5145	94.52
10	16	4	5094	94.19
10	16	8	5208	94.19
10	16	16	5562	94.19
10	32	4	5288	93.72
10	32	8	5371	93.72
10	32	16	5344	93.72
15	16	4	5237	93.92
15	16	8	5361	93.92
15	16	16	5707	93.92
15	32	4	5161	93.43
15	32	8	5169	93.43
15	32	16	5318	93.43

Table 1: garden.sif.y4m Encoding Results for Different Settings

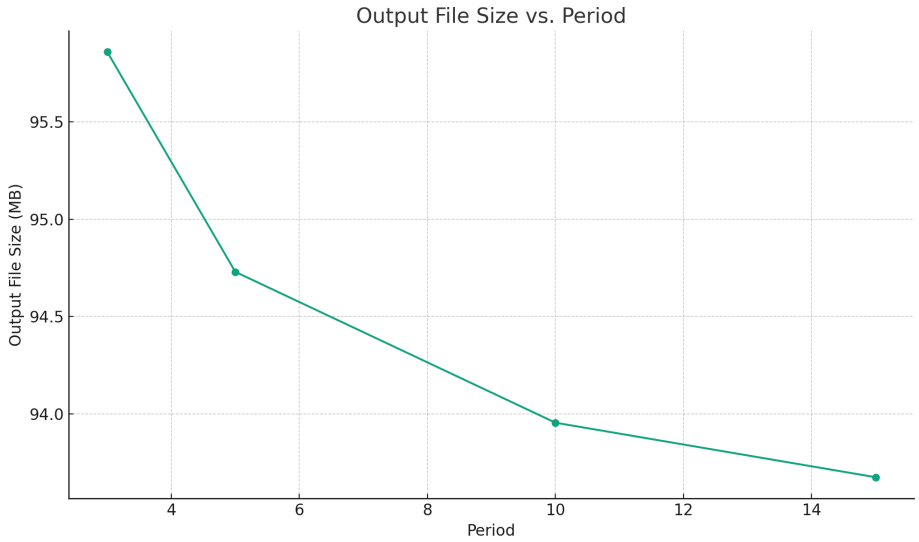

Figure 3: Output Size with Period

Regarding the decoder, the bar chart, Figure 4, compares the decoding times for three video files, each with a resolution of 176x144 pixels but differing frame rates. *deadline_qcif.y4m* has the highest frame rate at 1374 fps and also the longest decoding time, suggesting a correlation between frame rate and decoding complexity. *grandma_qcif.y4m*, with a frame rate of 840 fps, shows a moderately high decoding time. In contrast, *akiyo_qcif.y4m* with the lowest frame rate of 300 fps, has the shortest decoding time, further supporting that lower frame rates may lead to quicker decoding times.

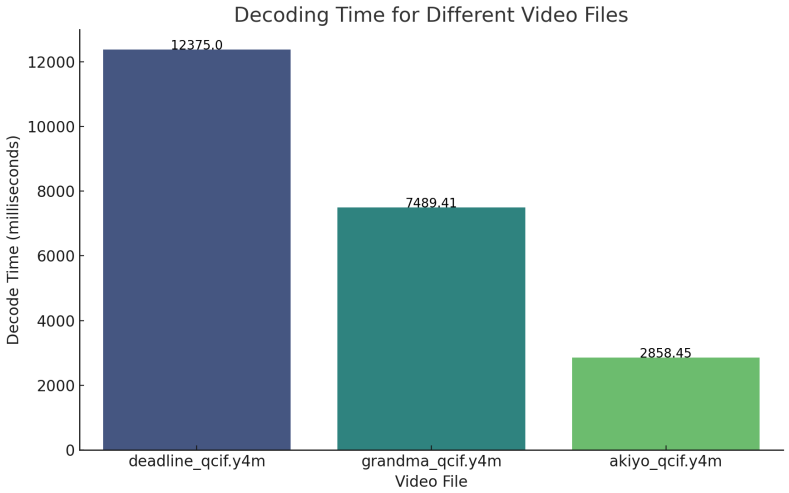

Figure 4: Different Video Files Decoding Time

4 Workload and General Information

The repository for the project is ⟨https://github.com/pedromonteiro01/ic⟩.
Each team member made an equitable contribution to the project's resolution.