DIGITAL PHOTO ALBUM USING EPIC

18-551 SPRING 2001

RUCHI MITTAL (rmittal),

JENNIFER WONG (jlwong)

Table of Contents

1. Introduction

1.1 Problem Description

Modern image-and-video-compression techniques offer the possibility to store or transmit the vast amount of data necessary to represent digital images and video in an efficient and robust way. Compressed image-and-video data is transmitted through the Internet with PCs. With the advances in VLSI technology, it is possible to access the Internet with networked multimedia applications on slim hosts such as personal digital assistants (PDAs) and cellular phones.

When people access the Internet, the most important thing for them is to obtain information. In this situation, applications like downloading pictures from the Web would require fast decoding. Sometimes a slight degradation would be acceptable due to high compression rates and fast decoding. Therefore, for slim hosts powered by batteries, the need to speed up the decoding of images without increasing power consumption is important.

One increasingly popular research area is face detection and recognition of faces in images. In such an application, the image should be of high enough clarity, so that this complex algorithm can be executed on it. Therefore an enhancement procedure should be available to help the user increase the clarity of images, together with a fast decoding algorithm.

Under these circumstances, for one implementation, we developed a convenient digital photo album. This photo album stores compressed image data. When needed, a data decompression algorithm decompresses these data, and a program that detects faces sorts out human faces.

1.2 Similar Previous 18-551 Work

algorithm used in this previous project is different from the EPIC decompression algorithm that we are using. Therefore, our project is significantly different from their project and other previous projects.

1.3 Overview of Solution

In this project, we will develop a digital photo album with three parts: image data compression/decompression, post-filtering, and face detection. This photo album will require stationary images (referred to from this point on as images) to be stored in a compressed form. When needed, the compressed images are decompressed. The Efficient Pyramid Image Coder (EPIC) algorithm implemented on the EVM executes these image data compression/decompression processes. Then, face detection will help search through the pictures to find the human faces, rather than the user having to look through all the pictures to find the ones he/she needs.

There will be cases when particular photos are degraded and blurry. So there should be an option that can allow chosen photos to be enhanced with post-filtering algorithms implemented on the EVM.

We will be focusing our attention and energy on selected aspects of this problem. We will implement the compression, decompression and optional post-filtering on the EVM. In addition, to make the demo more attractive, we will do face detection on the image with the PC.

2. Image Data Compression

Efficient Pyramid Image Coder (EPIC)[1] is an experimental image data compression utility written in the C programming language. The compression algorithms are based on a biorthogonal critically-sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. EPIC is designed for extremely fast

encoding and a slight degradation in compression quality (as compared to a good orthogonal wavelet decomposition).

In this project, EPIC encoder and decoder were implemented on a TMS320C6701 Evaluation Module (EVM) containing a Texas Instruments C6701 DSP (C67).

First, in this section, characteristics of EPIC are briefly explained from a theoretical point of view. Secondly, EPIC encoding (compression) and decoding (decompression) algorithms are introduced. Finally, the implementation of EPIC on the EVM and the results are reported.

2.1 EPIC

For data compression, a subband coding is an effective means. In the subband coding, original signals are recursively subdivided into subband signals. As one of the subband coding algorithms, Quadrature Mirror Filter (QMF) decompositions are popular since it has the following advantages. First, pyramids built with QMF kernels have a number of desirable properties for signal processing. Second, a hierarchical computation is highly efficient. Third, the number of coefficients in the transform is equal to the number of pixels in the original image.

One of the problems of practical QMF is that the QMFs require larger kernels. Especially, well-behaved kernels with an even number of taps tend to be rather large. This means that these filters require many floating-point multiplications. On the other hand, odd-tap kernels can be more compact than the commonly used even-tap kernels. To form an orthogonal basis set, these kernel must be applied on staggered grids. Kernels with a width as small as seven can produce highly accurate reconstructions. In transforms based on QMF kernels, the basis set is orthonormal. As a result, the sampling functions are identical to the basis functions. However, orthogonality is not required for image coding; it is only necessary that the transform be invertible.

they do form a linearly independent basis set. The only problem is that the filters violate the standard QMF criteria, and therefore a different set of filters must be designed for the encoding process.

The image vector **e** may be written as a weighted sum of basis vectors corresponding to shifted versions of the filters, which appears as columns in the matrix **F**. If the weighting coefficients from a vector **p** we have:

$$\mathbf{e} = \mathbf{F}\mathbf{p}$$

$$\mathbf{F} = \begin{bmatrix} 1 & & & & \\ 2 & -1 & & & \\ 1 & 2 & 1 & & \\ & -1 & 2 & -1 & \\ & & 1 & 2 & \\ & & & -1 & \ldots \end{bmatrix}$$

For encoding, we seek a matrix **G** that will deliver the coefficients when applied to the image. That is:

$$\mathbf{p} = \mathbf{G}^t \mathbf{e}$$

and thus

$$\mathbf{G} = (\mathbf{F}^{-1})^T$$

Then, the task is simple to invert **F**; the columns of **G** will correspond to the encoding filters. The true inverse sample function has no-zero taps over the entire image, but 15-tap sample functions work quite well in coding image. This 3-tap kernel is extremely easy to compute, using only shifts and adds, with no multiplications. They can be

2.2 EPIC encoding and decoding algorithm

2.2.1 Encoding (Compression)

Encoding consists of transformation, quantization and Entropy coding shown in Figure 1. The entropy coding consists of the Run-length and Huffman encoding, called variable length encoding (VLE).

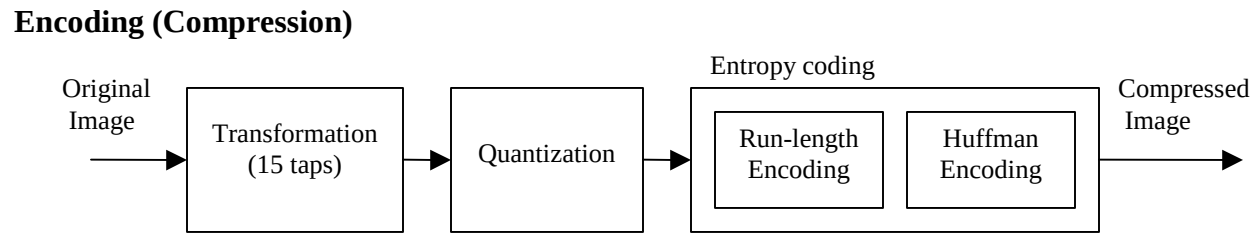


Figure 2-1: Block diagram of encoding

2.2.1.1 Transformation

The Quadrature Mirror Filter (QMF) *h(n)* (*n*=1,2,...,*L*), low pass filter, and its corresponding high pass filter *g(n)* (*n*=1,2,...,*L*) are used to implement the wavelet transforms, where L=15 in EPIC as described in 2.1.

Transformation process at a single level is shown in Figure 2-2. In order to transform 2-D image signals, two 1-D filters, *h(n)* and g*(n),* are applied to original image *x(n)* and downsampled by a factor of 2 in the X direction. Next, two filters are applied separately in the Y direction and also downsampled by a factor of 2 to produce H1, V1, D1, and L1. H1 contains horizontally oriented high frequency information, while V1 and D1 contain high frequency information oriented in the vertical and diagonal directions, respectively. L1 is a low pass filtered and downsampled version of the input signal. Then, since the L1 can be classified as most important and the other 'detail' bands can be

octave-band tree splits. The resulting four level subband decomposition is shown in Figure 2-3.

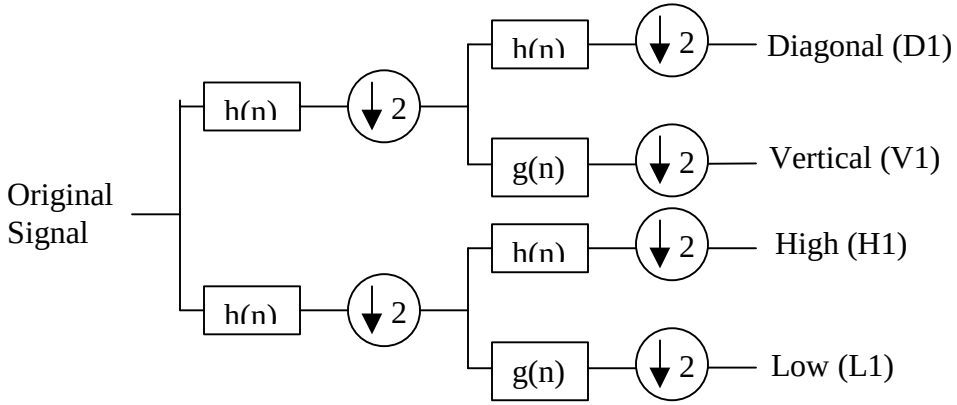
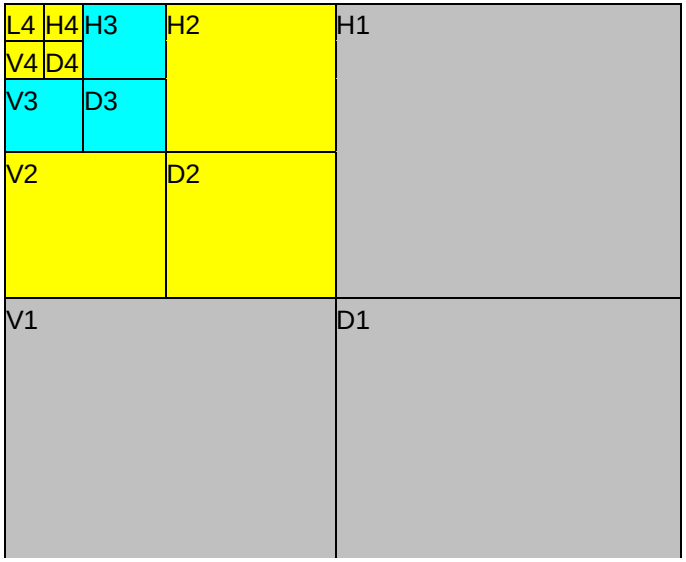


Figure 2-2: 2-D transformation at a single level



2.2.1.2 Quantization

Quantization refers to the process of approximating the continuous set of values in the image data with a finite (preferably small) set of values. The input to a quantizer is the coefficient transformed from original data, and the output is always one among a finite number of levels. The quantizer is a function whose set of output values is discrete, and usually finite. Obviously, this is a process of approximation, and a good quantizer is one that represents the original signal with minimum loss or distortion.

In EPIC, Quantization step is fixed for each band, decreasing by a constant factor at each level. Therefore, this is clearly not optimal.

2.2.1.3 Entropy Coding

In EPIC, entropy coding is composed of Run-length and Huffman coding. Entropy means the amount of information present in the data, and an entropy coder encodes the given set of symbols with the minimum number of bits required to represent them.

The Run-length coding assumes that the quantized coefficients at higher frequencies would likely be zero, thereby separating the non-zero and zero parts. The quantized coefficient is coded into a sequence of the run-level pair. The *run* is defined as the distance between two non-zero coefficients in the array. The *level* is the non-zero value immediately following a sequence of zeros. This coding method produces a compact representation of the coefficients, since a large number of the coefficients have been already quantized to zero value.

The Huffman coding finds the optimum (least rate) uniquely decodable, variable length entropy code associated with a set of events given their probabilities of

2.2.2 Decoding (Decompression)

Decoding algorithm performs the reverse operations of the encoding shown in Figure 2-4. As described in 2.1, at the inverse transformation, 3-tap [1 2 1] filter is used.

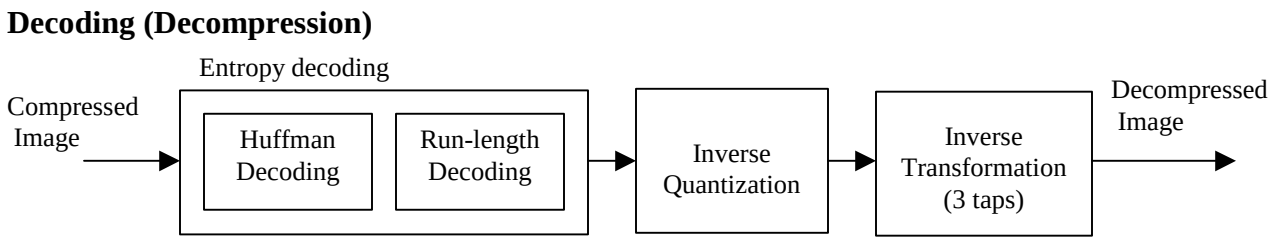


Figure 2-4: Block diagram of Decoding

2.2.3 Implementation on the EVM

As mentioned above, EPIC was implemented with C Program on UNIX, but only for 8 bits gray scale image data: Portable Gray Map (PGM) format. We implemented EPIC on the EVM as the following steps. First, the original algorithm for gray scale image was implemented on the EVM. Second, the program was modified to deal with color image. Third, the performance improvement in terms of speed was implemented.

In this project, we assume image size is 256 x 256 pixels for simplicity. Conversion from the original C program to the EVM implementation was based on the code from the Lab3. The implementation of the EPIC encoder and decoder on the EVM are shown in Figure 2-5, Figure 2-6.

At the encoder, the PC will read image data and transfer it to the EVM. Then, on the EVM, transformation, quantization and Run-length encoding will be applied to the

EVM and then these signals will be transferred to the PC. Finally, the PC will concatenate together and add the EPIC header and some parameters such us image size to the data and write the compressed image data into a file.

At the decoder, first, the PC will read compress image data. Then transfer low image data to the EVM. After performing Huffman decoding for high frequency images, these decoded signals will transfer to the EVM. On the EVM, Run-length decoding will be applied to this signal, then inverse quantization and inverse transformation will be applied to the all data. The EVM will transfer the decoded data to the PC. Finally, the PC will write the decoded signal to a file in PGM format.

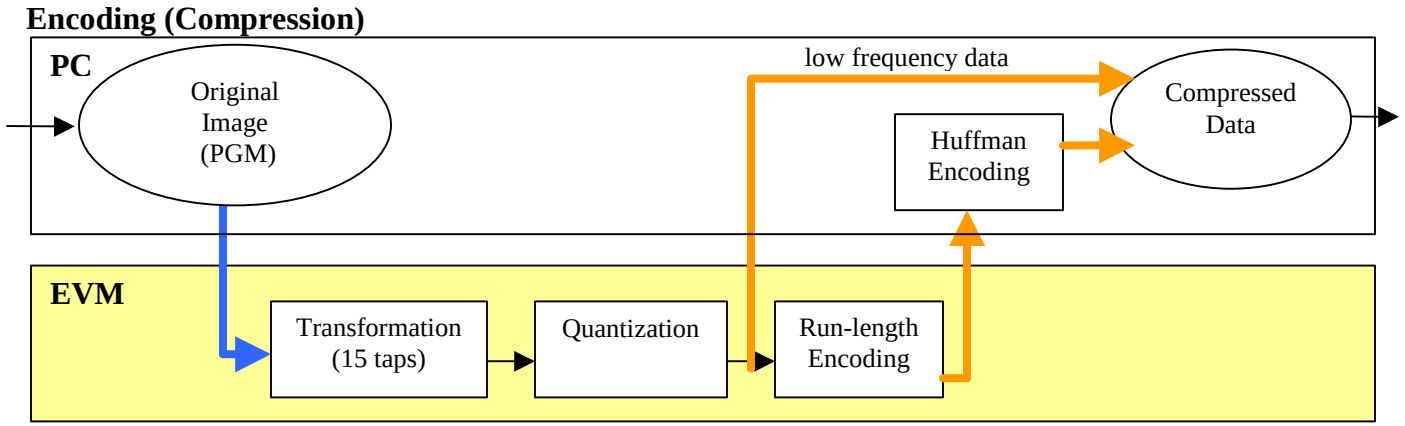


Figure 2-5: Implementation of the encoder on the EVM

Figure 2-6: Implementation of the decoder on the EVM

2.2.3.1 Color image

In order to correspond to color image, we assume color image is given by Red-Green-Blue (RGB) data in Portable Pixel Map (PPM) format. Under this assumption, implementing color image was straightforward – processing each of the color images separately. In the PC program, color image or gray scale image will be automatically detected by reading header file in PGM or PPM file.

2.2.4 Result

Figure 2-7, Figure 2-8 shows the comparison between the original image and the compressed image for gray scale and color images respectively.

The original “Lena” image
at 256 x 256 pixels

Compressed image with EPIC
Compression ratio: 1/20

The original “GIRL” image
at 256 x 256 pixels

196666 bytes
(8x3(RGB)=24 bits per pixel)

Compressed image with EPIC
Compression ratio: 1/20

18789 bytes (2.3 bits/pixel)

Figure 2-8: Color image

The encoding and decoding program are stored in the on-chip memory; on the other hand, image data was stored in the 8MB external memory on the EVM. Table 2-1, Table 2-2 show the EVM Memory usage for the Encoder and the Decoder respectively.

Table 2-1: EVM Memory Usage for the Encoder

Encoder	
Image	256*256*4bytes
Transformed signal	256*256*2bytes
Quantized step data	13*2bytes
Entropy coded symbol stream	256*256*2bytes

Table 2-2: EVM Memory Usage for the Decoder

Decoder	
Compressed data	256*256*2bytes
Unquantized data	256*256*4bytes
Quantized step data	13*2bytes
Entropy coded symbol stream	256*256*2bytes
Transformation filter	0 (shifts and adds)
Decompressed data	256*256*4bytes
Total	786Kbytes

Table 2-3 shows the amount of time for each process of the encoder and decoder. The transformation of the encoder takes 233.0 Million cycles, 12 times longer than inverse transformation of the decoder for the original version, since encoding filter is 15 taps.

Table 2-3: Cycles for encoding/decoding on EVM

Encoder	Million cycles	Decoder	Million cycles
Transformation	233.0	Inverse Transformation	19.0
Quantization	31.4	Inverse Quantization	6.1
Run-length	2.8	Run-length	1.4

2.2.4.1 Performance enhancement

From Table 2-3, our performance improvement efforts focused on the

functions is not easily to be modified, we applied the loop-unrolling only to the center function. After implementing the loop-unrolling, the cycles decreased to 128.4 Million cycles. We got the big improvement. Since the epic transformation function was not optimized, we optimized a lot while implementing the loop-unrolling. Therefore, the reason for this big improvement is not only the loop-unrolling, but also the code optimizing by hand.

Next, we implemented the paging on the loop-unrolling program. The paging was also applied to the center function of the transformation and the workspace size in on-chip memory is 256 (pixels) * 24 (lines) * 4 (bytes) = 24.5Kbytes. After implementing the paging, the cycles decreased to 104.1 million cycles.

Table 2-4 Cycles for encoding/decoding transformation

		Million cycles
Encoder	Original	233.0
	Loop unrolling	128.4
	Paging	104.1

Inverse transformation has little room for the parallelization because it has already optimized. In addition, since inverse transformation process has lack of memory locality, it was difficult to implement paging. Due to time constraint, we did not improve the inverse transformation.

3. Image Enhancement [2]

EPIC Compression is a lossy algorithm, meaning that there is going to be loss of data. Hence, the original image cannot be fully restored. However there are numerous techniques that can be used to reduce or remove artifacts present after decompression. EPIC's main advantage over other compression algorithms is that it is fast. We therefore decided to filter the digital image. We used the low pass filter to help make the image more aesthetically pleasing, and the high pass filter for edge enhancement. The high pass would have been useful for enhancing facial features for face detection purposes.

3.1 Smoothing Filters

Smoothing filters are designed to reduce the noise, detail, or 'blurring' in an image. Typical smoothing filters perform some form of moving window operation that may be a convolution in the window. It is easy to smooth out an image, but the basic problem of smoothing filters is how to do this without blurring out the interesting features. For this reason, much emphasis in smoothing is on 'edge-preserving smoothing'.

3.2 Edge Enhancement Filters

Edge enhancement filters are the opposite of smoothing filters. Whereas smoothing filters are lowpass filters, edge enhancement filters are high pass filters and their effect is to enhance or boost edges.

We sampled many different filters, and found that these 2 had the best results.

1) Smoothing Filter

1 2 1

2) Edge Enhancement Filter

-1/3	-1/3	-1/3
-1/3	11/3	-1/3
-1/3	-1/3	-1/3

We tested filters using the MATLAB command ‘filter2’ on an array of image values. The electronic copy of the entire code is attached.

e.g.

```
fname=sprintf('lena.raw', i);
  if(fid > 0)
        input(:,:) = fread(fid, [256 256], ‘uint8’)';
    fclose(fid);
    disp(['Read file ' fname]);
  else
    input(:,:) = zeros(256, 256);
    disp(['File ' fname ' not found']);
  end
end

H = [1 1 1 ; 1 1 1; 1 1 1]/9
I = filter2(H,input)/255
Imshow(I), figure; Imshow(input)
```

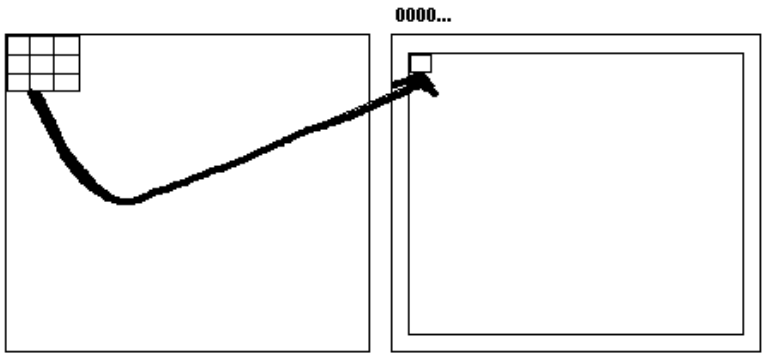


Figure 3-1: Spatial Correlation performed on input image

3.4 Improving Speed Performance

Without any improvements, the algorithm took 3.5 million cycles, where each cycle is 40ns. After implementing paging, DMA transfers and loop unrolling, this decreased to 0.49 million cycles.

However, the external memory interface (EMIF) on the C6701 processor is very slow. Single word accesses to external memory take roughly 15-17 cycles per access. Accesses to the internal (on chip) RAM only take 1 cycle. This filtering procedure accesses each pixel 9 times. So if external memory is storing the image than each pixels would need 15*9 = 135 cycles to compute. It would be faster if each pixel was first read from external memory (15 cycles) and 9 times from On-Chip for a total of 24 cycles. A problem with using On-Chip is that it is limited in the fact that internal data memory is much less than external, being only 64K. This won't be large enough to hold an entire image. So we needed to copy blocks of the image one at a time into internal memory and calculate those results. We calculated the output for blocks of 16 lines.

Since there is a large overhead only for the first transfer of each pixel, it would be

Our final speed improvement implementation was loop-unrolling. This was useful since our program contained nested loops, as the optimizing C compiler only pipelines the innermost loop.

3.5 Results of image enhancement

The smoothing algorithm worked well with faces, as it reduced any harsh artifacts and lines present. It didn't work as well with other images like boats. The ship name became too blurred. The edge enhancement algorithm also worked well with facial images. The iris of the eyes was clearer, which would help the face detection process.

Here are some examples of the images prior to and after filtering.

Image after compression **Image after filtering**

Figure 3-2: Filtering result

4 Face Detection

recognition. Once the face is detected, the digital camera can apply separate recognition algorithms to recognize the face. This will help us simplify the process of sorting through our digital photo album. Originally, we had wanted to try to incorporate face recognition into our project if we had time after finishing face detection. However, because of time constraints, we could only focus on the detecting aspect of face recognition in our project demo.

We choose to include face detection in the project in order to make the lab demonstration more interesting. We did not intend to make face detection a real focus of the project. As a result, we only wanted to run face detection on the pc based on time constraints and memory constraints on the computer.

4.2 Software

4.2.1 Software Background

The face detection is done using the Face Detection software developed by Henry Rowley, an alumni of CMU[3]. We had difficulty obtaining this code in its original version because we could not contact anyone that had access to this code. The closest code that we could obtain to Rowley's face detection software was a face tracking software program that contains Rowley's face detection software. The tracking software was developed by Professor Tsuhan Chen and his team. We were able to contact Professor Chen and obtain a copy of the face tracking software that he and his team created.

4.2.2 Face Detection

Rowley's face detection code uses neural networks and training to find faces in an gray scale input image. Rowley's program utilizes a face detection neural network to

detectors, an arbitrator pieces the results from each individual detector and then determines if one or more faces are present in the input image.

4.2.2.1 Neural Network Used in Rowley's Face Detection software

The neural network detectors use a process called "face labeling" when searching for a face in an image. With face labeling distinguishing points on the face such as the pupils in the eye, the nostrils of the nose, and the corners of the mouth are selected and marked on an image. The individual detectors attempt to locate these points on an input image. If the points are present the detector returns a positive result indicating that a face has been found.

4.2.2.2 Algorithm for Detection Process

In Rowley's code, the face detection neural network receives a 20 x 20 pixel region of the input image. This region is preprocessed using lighting correction and histogram equalization. Following this it is passed on to the neural network detectors. The neural network runs detectors all over this 20 x 20 pixel input image. The results are returned and processed determining how many faces are present in this input window. If a face is larger than this window, the input image is repeatedly reduced in size by a factor of 1.2 using subsampling. Detectors are run on the preprocessed, subsampled image at each size. Following this, the arbitrator unit pieces the results of each detector and then determines the number of faces present in the image. We have included a graphical version of the basic algorithm for detection, obtained from Henry Rowley's thesis report.

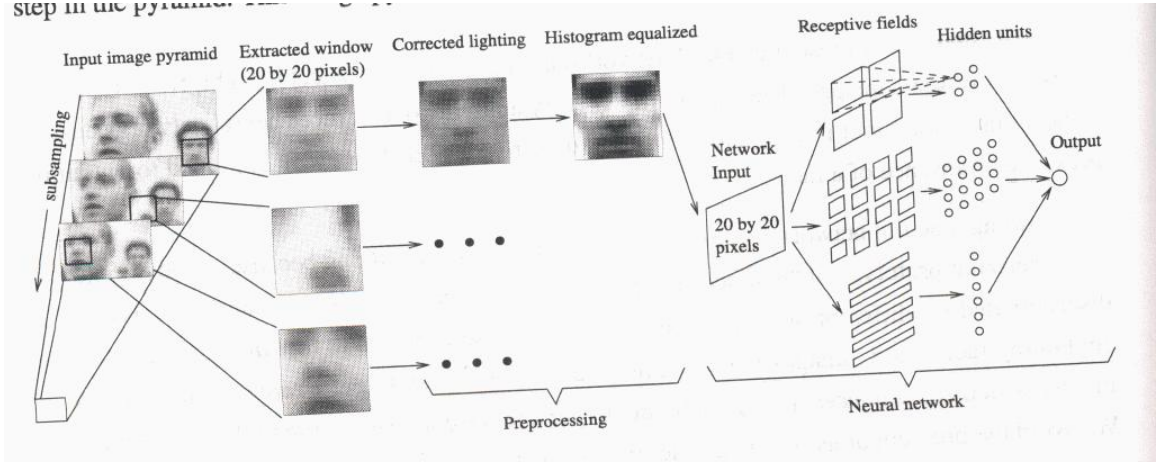


Figure 4-1: The basic algorithm used for face detection

4.2.3 Face Tracking

Face detection is the initial step to another process called face tracking. Face tracking is the process of locating and following a face in a video clip. The face tracking software basically accepts a color video clip as an input. It then takes the first frame from this video clip converts the frame into grayscale, and then runs face detection on it. Once the location of the face or faces have been found, this process is repeated on the next succeeding frame in the video clip. The face is tracked within the video by piecing together where the face has moved in the new frame relative to the old frame.

4.3 Assumptions

There are several challenges in original face detection that we must note. For instance, problems could arise with variations in the size of the face, angle of rotation of the face (face is frontal but is tilted sideways some angle), brightness and contrast of image (attributed from changes in light source), variations on the subject's pose (face is

image during detection. Lastly, facial expressions also had to be accounted for. This can include closed or opened eyes and mouth and whether a person is smiling or frowning.

In order to simplify the task of face detection, and to reduce the size and the complexity of the code, we did not account for many of the problems mentioned above. The face detection algorithm that we used made several assumptions about the image that it is applied on. We only used images where the subject's face is in a frontal position, the face is not tilted sideways, and the face is not tilted up or down too much. The facial expression of the person was restricted to a person with a closed non-smiling mouth and open eyes. The face detecting algorithm was only used on images that vary slightly in brightness and contrast from the given sample face. The images used had a very uniform one-color background that was easily distinguishable from skin color, similar to a mug shot. The images we used followed these characteristics. We used the face detection algorithm on this image scaled at a larger and a smaller size. We also had a few images that did not have any faces present in them, so that we could show that the program mentions that there is no face present, when a face cannot be found in the image.

4.4 Modifications

4.4.1 Problems with Original Code

We had several problems with the original code that we obtained from Professor Chen's. Professor Chen's software is actually for use on face tracking. As explained above, the program takes a video clip as an input and then runs face detection on each individual frame of that clip to perform tracking. Because we only need to perform face detection on an individual still frame we did not need much of the extra code that is used for tracking. As a result of the additional tracking code, the face tracking program

time of face detection on the input image much faster. Following this, we also needed to modify the input to this code. As discussed before, the original code accepts a video clip as an input. We needed to perform face detection on a single gray scale image. As a result, we modified the parameters of our new face detection code so that it can accommodate this. After all these modifications the code should work such that when it receives an input image, it should determine whether or not a face is present. If a face is present it should mark an "x" on the eyes and then place a box on the face region. This can be seen in the following figure (Figure 4-1).

Image before detection **Image after detection**

Figure 4-2: **Desired result for image with face present**

If a face is not present it should also return a response indicating so. The following figure is an example of this (Figure 4-2).

Image before detection **Image after detection**

Figure 4-3: Desired result for image without face present

After we finished the face detection modifications, we integrated the EPIC compression/decompression code and the image enhancement code together with the face detection, so as to get the entire project working together.

4.5 Final State

After integrating the different parts of the project, we tried to run the final project so that we could get our results. However, we had a few linking errors when we tried to run face detection together with the other parts. We really tried to examine these errors and try to fix them. One of the TA's for this class, and one of Professor Chen's students,

attaching it to another program. As a result, we were unable to test the final detection code, or to test the entire program integrated together. Since the face detection part of the project was not the main purpose of the project, we spent more time trying to perfect the other parts, and so we were not able to spend a lot of time trying to figure out why we were getting the errors for the face detection part. The actual face detection program consists of several sub-programs. However, a lot of these programs could not be modified. We basically modified only one of the main programs, and so we have attached an electronic copy of that sub-program. If needed, the entire program can be obtained from us.

5. Conclusion

As we can see from the project described above, the Digital Photo Album using EPIC would be a convenient application, owing to its ability to help the user save a lot of space by storing the image in compressed form. Since EPIC is a fast decoding algorithm, as we have seen, the user will not spend a lot of time while the program is decompressing the compressed images saved on the PC. There is some noticeable loss in the clarity of the image, which can be reduced by filtering the image. We also applied a face detection algorithm to the image, so that it would be able to detect a human face in the image, and sort out the image according to the presence of a face in them. This would help easily organize the digital photo album. A further step that could also be applied to this project is face recognition, which would help recognize the face of the person in the image. However, due to time constraints, we did not implement this part of the project.