Scott Lembcke

Lossy Image Bisection

For my project, I applied using image bisection in a lossy manner for image and video compression. The basic idea is that each channel in the image is recursively split in half along its longest axis and any node that contains values that differ by less than a given threshold are replaced by the average of all the values. The result of this process is a binary tree with the image information stored in the leaves. This tree can then be serialized and stored. I chose to do so by storing the tree structure using a prefix code, and then following that by storing the leaves in order of their occurrence in the tree. One benefit of using this format is that it is possible to traverse the tree without actually reconstructing it. As a final step, I would compress the serialized tree with gzip. To implement my algorithm, I created a number of command line programs that use standard input and output to create a pipeline. Several stages were possible. When compressing, one program would read a PNG file and decompress it to a raw 8 bit RGB stream. This would then be passed to a program that would separate the color channels. I wrote two such programs, one that would output RGB, and another that would output YCbCr. Eventually I exclusively used the YCbCr program, as it allowed roughly twice the compression with similar output quality. After the channels were separated, they are passed to the compression program which bisects the image and outputs the serialized tree. Finally, this tree is passed to gzip as a final compression step. Decompression proceeds much the same way, the gzipped tree is decompressed, and used to generate the image channels, which are then composed into RGB and written to a PNG file.

Video compression simply works by compressing the differences between the last compressed frame and the next frame. That way it doesn't accumulate any more error than the amount used for the bisection tolerance.

The core programs were implemented in C that used stdin/stdout, a ruby script was used to construct the pipeline. I had never really created a program using a pipeline in this way, but I found it to work very well for this project. This was especially useful when I got to video compression. I realized that all I had to do was to modify the pipeline by adding a few more stages. My implementation uses, but does not depend on, libPNG and the gzip program. libPNG was used for reading and writing .png files, while gzip was used to further increase the compression ratio of my program. The source can be found at http://epoxy.morris.umn.edu/~lembckesd/results/src.

My algorithm works best (compression ratio wise) on large images with low frequency information in them, and worst with small images with high frequency information. The best case is that an entire image could be replaced by a single tree node, allowing the image to be compressed to a few bytes. The worst case would be when every pixel has its own tree node, this means that every pixel value would be present in the output along with the tree structure. The tree structure is approximately 2n bits long, where n is the number of nodes in the tree. So in the worst case, where p is the number of pixels, you would have approximately p + (2p)/8 bytes, or a 25% expansion. When working with video, it performs best when a large portion of the frame that changes by a constant, as a large solid colored region can usually be replaced by a small number of bisection boxes. The worst case is when the difference has a large area of erratic changes, as this becomes a lot of high frequency

information to encode.
The images and movies in the results can be found at:
http://epoxy.mrs.umn.edu/~lembckesd/results

Image Compression Ratios: (without zlib / with zlib)

tol \ img	**clouds.png**	**lena.png**	**mandril.png**
0	1.7 / 3.3	0.85 / 1.4	0.89 / 1.3
8	35 / 43	2.2 / 2.7	2.0 / 2.4
12	72 / 83	3.8 / 4.5	2.7 / 3.0
16	116 / 135	5.4 / 6.2	3.2 / 3.6
24	281 / 305	11 / 11	4.7 / 5.1
48	1721 / 1787	33 / 34	9.7 / 10
64	4562 / 4546	57 / 59	14 / 15
96	21253 / 20045	150 / 153	33 / 34

Lossless Image Compression Comparison:

	Bisect	**PNG**
clouds.png:	3.33	2.86
lena.png:	1.35	1.54
mandril.png:	1.33	1.29

Lossless Animation Compression Comparison:

Mine:	161
8bit RLE	55
1bit RLE	216

Video Compression Comparison:

H.264	270
Bisect (tol 24)	77
Bisect (tol 12)	37

From the image compression results, you can see that the cloud.png picture compressed the best as it had a lot of low frequency information in it. For this reason, it also becomes unrecognizable at a much lower tolerance than the other two pictures. The other two pictures didn't perform nearly as well, but still increased the compression ratios over PNG at low tolerances without introducing highly visible artifacts. On the clouds.png and mandril.png images, lossless bisection beat PNG's compression. I would guess that lena.png didn't show the same results due to the

noticeable amount of image noise. Also interesting is that although bisection increases the size of images at low tolerances the final zlib compressed file is smaller than the PNG file it came from (which also uses zlib). I would guess this is because the tree provides better entropy coding, placing more adjacent pixels near each other in the output.

The lossless animation compression is where my algorithm really shines. The animation used was one of my black and white physics animations. This allows very large areas to be encoded in very few tree nodes. Also, because it's black and white, there is little entropy in the luminance, and none in the chroma. Compressing the animation as a 24bit color even beat Quicktime's 8bit grayscale RLE codec. However, it couldn't beat the 1bit RLE codec. Both RLE codecs use frame differencing to increase the compression.

The video I choose to compress didn't fare nearly as well. Looking at the frame differences, there is simply too much change between frames to get good compression. Compared to the state-of-the-art H.264 video compression codec, image bisection looks pretty abysmal. Providing video with very noticeable artifacts and relatively poor image compression ratios.

Further improvements to my algorithm could include compression of the tree structure which probably contains a lot of redundant subtrees. Another possibility would be a better representation of tree nodes. Rather than simply storing the average color, some sort of transform could be done to increase the quality of the nodes.