An Optimizing Lossy Generalization of LZW

Steven Pigeon
Université de Montréal
pigeon@iro.umontreal.ca

Abstract

We present a lossy generalization of Welch's LZW algorithm. We first discuss two lossy generalizations of LZW. The two variations either try to maximise compression without worrying about image quality, or try to maximise image quality without worring too much about compression. We then present a new algorithm, also lossy, which optimizes compression according to an objective function supplied by the user, that enable him to balance between compression ratio and image quality.

1 Why a Lossy Generalization of LZW?

LZW [Welch84] is an interesting algorithm. It is simple to implement, fast when well implemented, and is present in many standards and commercial products. For example, LZW is the compression algorithm used in Compuserve's GIF87a/89a image format [Comp89], which in turn is supported by web browsers and countless image processing software. While it may be critical for some expensively acquired images (such as deep space probe data) to be compressed losslessly, it makes little sense for most other images.

Others have suggested using palette reduction as a mean of reducing an image's size, but this technique must be rejected. The idea behing palette reduction is the following. Say we start from a 256 entry palette, and we go down to 64 entries. We should benefit from a 25% reduction of file size, plus from any repetitions that would now occur as a result of the reduction of palette: the probability that two adjacent pixels have the same value should be higher than it was before. But it is not so. First, by reducing severy the image palette size, we introduce important color aberration, secondly, this reduction in the number of distinct colors calls for a mean of compensation, and it is usually dithering. While dithering may be visually adequate, it fouls up compression, since patterns that would be repeating are contaminated with what appears to be random noise. This noise is the result of the error propagation dithering algorithm.

We proposed in 1995 a variant [Pigeon95] that searches greedily in its dictionary for possible matches. Another lossy algorithm was proposed by Chiang in [Chiang98]. In basic LZW, one searches for the longest string in the dictionary that exactly match the input. In an image compression application, the dictionary contains strings of pixels rather than, say, strings of letters. In the particular case of GIF image compression, the pixels are in fact $1 \leq n \leq 8$ bits integers pointing into a look-up table of at most 256, 24 bits colors. Entries

in the dictionary are therefore strings of these integers. While in basic LZW, only exact matches are allowed, in the Pigeon and Chiang variants, differences between input and a dictionary entry are acceptable. One can use a pixel instead of another if the color difference is within a specified tolerance. Neither algorithms optimize explicitely for compression nor for image quality. They can lead to both bad compression and bad image quality. In this short paper, we present another variant that choose matching strings in the dictionary in a way that both optimize for compression and image quality.

The algorithms we describe use a metric that defines the similarity between two symbols or two strings of symbols. The metric used depends on the nature of the symbols forming strings. If, as in our case, the strings are pixels (tripplets of the form (r, g, b), with $r, g, b \in \mathbb{Z}_n$) the metric will be some color difference mesure, for e.g.

$$\|a - b\|_m = \sqrt{\gamma_r^2(a_r - b_r)^2 + \gamma_g^2(a_g - b_g)^2 + \gamma_b^2(a_b - b_b)^2} \tag{1}$$

where $\gamma_r \approx 0.299$, $\gamma_g \approx 0.587$, $\gamma_b \approx 0.114$, for two pixels a and b. In eq. (1) a_r stands for the red component of pixel a, a_g for its green component and a_b for its blue component. Eq. (1) is considered a good *linear* approximation of the eye's response to color differences in terms of brightness. For strings of pixels, we can define the color difference to be

$$\|s - w\|_m = \sum_{i=1}^{|s|} \|s_i - w_i\|_m \tag{2}$$

where s_i stands for the i-th pixel of string s. This metric gives the total perceptual brightness difference between strings of pixels. More sophisticated metrics can be used, such as metrics based on other color spaces or a function based on MacAdam's just noticeable color differences (also known as MacAdam's ellipses [MacAdam42]). The metric is important because it controls the compressed image quality.

2 The Original LZW algorithm

The original LZW algorithm was proposed in [Welch84]. This algorithm processes its input sequentially, from first to last symbol. The input is compared to all the strings contained in the dictionary at that time. The string in the dictionary that match the *longest* part of the input is chosen. The matching string is removed from the input and a code representing its index into the dictionary is output in its place. A new string, formed with the last matching string and the first input symbol that did not match is added to the dictionary. The process repeats until all symbols are processed. This scheme permits compression if the strings that match are significatively longer than the index codes. Adaptive coding strategies that take into account index probabilities have lead to better compression than GIF87a/89a scheme where code length is fixed to $\lceil \lg |D_t| \rceil$

(the number of bits necessary to represent any number in $0, 1, \ldots, |D_t| - 1$).

While the *efficient* implementation of such as compression scheme is worth discussion, it is not the object of the present paper. We will use an notation to describe the operations of searching in the dictionary, updating the dictionary and updating position in the input, but hides the internals of each operation. We will not concern ourselves with coding. Let the shortcut notation I_a^b represent the input segment $\{I_a, I_{a+1}, \ldots, I_b\}$. Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$ be our symbol alphabet, and $|\Sigma|$ the number of distinct symbols in the alphabet. Let D_t be the dictionary state at time t and $p(t)$ the position in the input at time t. At $t = 0$, the dictionary contains $|\Sigma|$ strings, each one containing a different symbol σ_i. This is ensure that a segment of the input always match something in the dictionary, even if it is a degenerate string of length one. This removes the need for an escape code to introduce new symbols into the dictionary when they are first met in the input. The position $p(0)$ points to the first symbol of the input. The string that is chosen from the dictionary at time t is given by

$$\hat{s} = \text{argmax}_{\left\{ s \in D_t \,\middle|\, s = I_{p(t)}^{p(t)+|s|-1} \right\}} |s| \tag{3}$$

which is only the longest string in the dictionary that matches the input. The dictionary is updated by

$$D_{t+1} = D_t \cup \{\hat{s} : c\} \tag{4}$$

where $c = I_{p(t)+|\hat{s}|}$ and $a : b$ stands for concatenation. Update is made by adding a new string to the dictionary. The new string is the last matched string concatenated with the first symbol that did not match. This symbol prevented any string in D_t to match the input; we are sure that $\hat{s} : c$ is not already in the dictionary, and we add it in the dictionary, since it is part of the input, so it can be used if encountered again. The position is updated by

$$p(t + 1) = p(t) + |\hat{s}| \tag{5}$$

In the compressed stream, a code for $\hat{s}$ is emitted. During decompression, the indexes are read from the compressed stream one by one. A string s_t is found at time t indexed by the code read from the compressed stream. This string s_t is output. The index for the next string, s_{t+1} is decoded. The dictionary is then updated by

$$D_{t+1} = D_t \cup \{s_t : first(s_{t+1})\}$$

where $first(w)$ returns the first symbol of the string w, after what s_{t+1} is ouput. This process is repeated until no more indexes can be read.

3 Pigeon and Chiang Variants

In these variants, rather than looking for the longest exact match between the dictionary and the input, we will be looking for a *tolerable matches* under a given

metric m. In Chiang's variant, that he calls LLZW, a string matches if all of its symbols are tolerably different than the input under the metric m. Tolerance is controled by a possibly adaptive threshold value, say θ, set by the user. Under Chiang's variant, any θ-tolerable string in the dictionary is a possible match. A string s is said to be θ-tolerable to a string w iff

$$\|s - w\|_m \leq \theta K$$

for some constant K. Chiang suggest to chose K such that it is proportional to the (brightness) variance of the current local region of the image. Eq. 3 is replaced by

$$\hat{s} = \text{argmax}_{\left\{ s \in D_t \,\middle|\, \|s - I_{p(t)}^{p(t)+|s|-1}\|_m \leq \theta K \right\}} |s| \tag{6}$$

Chiang's variant picks the longest θ-tolerable string in the dictionary. This can be desastrous to image quality, since the longest matches can be quite far from the best quality matches, which in turn are more likely to be shorter. In Pigeon's variant, a string is said to be ϕ-tolerable iff

$$\bigwedge_{i=1}^{|s|} \left(\|s_i - w_i\| \leq \phi \right) \tag{7}$$

However, the data structure used in [Pigeon95] being a trie, each symbol is matched one at a time as the algorithm walks along the trie. A trie is simply a k-ary tree, in which each path from the root to a node represent a string in the dictionary. Fig. 3 shows such a trie. The algorithm goes down the trie by going to the next reachable node that have the best score. At input position $p(t)+i$, we are positioned on a node of depth i. This current node (determined by previous decisions) can be either a leaf or have up to $|\Sigma|$ sons. If it's a leaf, then it represent a ϕ-tolerable string, and this string is the match. If it is not a leaf, the next node to explore is chosen as the node containing the symbol having the smallest difference under metric m with the input. Let T_i be the current node and $T_{i,j}$ its j-th son. The next match is given by $T_{i+1} = \text{argmin}_{T_{i,j}} \|I_{p(t)+i+1} - T_{i,j}\|_m$. If T_{i+1} is ϕ-tolerable, the algorithm continues to search the trie at node T_{i+1}. If T_i has no ϕ-tolerable son, the match stops at T_i.

As it proceeds in a greedy fashion, the algorithm is prevented from discovering globaly longer and/or less noisy strings that may hide behind a localy bad but ϕ-tolerable score. This limits compression since the strings that have a better quality are likely to be shorter than the longest ϕ-tolerable string in the dictionary, while it gives better image quality for $|s|\phi$ comparable in magnitude to θ.

4 An Optimizing Variant

Chiang's and Pigeon's variants either imperil compression ratio or image quality. A better algorithm would be one that tries to maximize some objective function

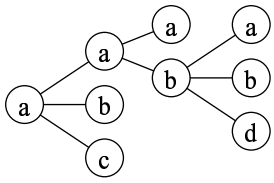

Figure 1: Trie containing strings a, aa, aaa, aab, $aaba$, $aabb$, $aabd$, ab and ac, therefore representing the dictionary $\{a, aa, aaa, aab, aaba, aabb, aabd, ab, ac\}$. Each node can end a string. The total number of representable strings equals the number of nodes in the trie.

over the dictionary. Let call this algorithm P-LLZW to distinguish it from Chiang's LLZW and Pigeon's first variant, G-LLZW (for *greedy* LLZW). The objective function in algorithm P-LLZW should compute an objective score that takes into account both compression ratio and image quality. Let $f_{m,\phi}(s, w)$ be such an objective function that takes into account both compression and image quality. Eq. 3 now becomes

$$\hat{s} = \underset{s \in D_t}{\operatorname{argmax}}\ f_{m,\phi}(s, I_{p(t)}^{p(t)+|s|-1}) \tag{8}$$

Dictionary is still updated according to eq. 4 and position to eq. 5. Eq. 8 is solved by dynamic programming or another efficient technique such as minimax search, depending on the data structure holding the dictionary and the nature of $f_{m,\phi}$. You may wish to have $f_{m,\phi}$ such that

$$f_{m,\phi}(s{:}c, w{:}d) = f_{m,\phi}(s, w) + \omega f_{m,\phi}(c, d) \tag{9}$$

so the optimality principle applies and that dynamic programming and other search methods are possible. The objective function $f_{m,\phi}$ must be chosen carefully to get interesting results. It can be a combination of the form

$$f_{m,\phi}(s, w) = \lambda_1 g_{D_t}(s) - \lambda_2 h_{m,\phi}(s, w)$$

for a weighting constants λ_1 and λ_2. The function $g_{D_t}(s)$ gives a score according to the length of the match, that is, the compression ratio. This would involve the length of the code that would be output for s at time t, which in turn depends on the state of the dictionary at time t, D_t. The function $h_{m,\phi}(s, w)$ uses the metric m and the threshold ϕ to compute a quality score. $h_{m,\phi}(s, w)$ is 0 when the match is perfect ($s = w$), and is infinity when the match is not ϕ-tolerable. This will ensure that the algorithm finds only ϕ-tolerable matches in the dictionary. However, the rate at which $h_{m,\phi}(s, w)$ goes to infinity can be in such a way to allow "soft ϕ-tolerance", by which a string that is $(\phi + \varepsilon)$-tolerable does not have an infinite penalty associated to it but only somewhat greater than if it were only ϕ-tolerable.

5 Results of Algorithm P-LLZW

We used a trie as the dictionary data structure, as in earlier implementations, and we used a depth-first search in the trie, pruning only when the threshold ϕ

was exceeded (as in eq. 7). This leads to an algorithm whose worse case is linear in the total number of symbols contained in the dictionary. We tried various objective functions:

$$f'_{m,\phi}(s,w) = \lambda|s| - (1-\lambda)\|s-w\|_m$$
$$f''_{m,\phi}(s,w) = \lambda\frac{|s|}{\lg|D_t|} - (1-\lambda)\|s-w\|_m$$
$$\vdots$$

for various λ. However, we chose

$$f_{m,\phi}(s,w) = \lambda|s|^2 - (1-\lambda)L\|s-w\|_m \tag{10}$$

where L is a constant that depends on the metric m and $\lambda = \frac{1}{2}$. We used the metric described in eq. (1) and eq. (2), assuming that each of the components varied from 0 to 255 and this gives $L = 1$ in eq. (10).

Results are given in Table 1. With objective function eq. (10), it is the parameter ϕ that control the compression quality and ratio. The signal to noise ratio (SNR) was computed using the metric and the following formula

$$SNR(s,\hat{s}) = 10\log_{10}\frac{\sum_{i=1}^{|s|}(\gamma_r s_i(r) + \gamma_g s_i(g) + \gamma_b s_i(b))^2}{\sum_{i=1}^{|s|}\|s_i - \hat{s}_i\|_m^2}$$

where here s and $\hat{s}$ represent the whole original image and the whole reconstructed image, respectively. The SNR of various images give an indication on how colors and brightness are preserved in the P-LLZW compressed images. A SNR above 30 dB is considered good. Above 40 dB, it is considered excellent. Under 25 dB serious degradations are visible. As one can see from table 1, P-LLZW gives some surprisingly good results, as with image "P-Boxes" where the BPP goes from 3.7 to 1.4 while maintaining a SNR of 34 dB. Reducing the bit rate by one bit per pixel gives excellent image quality, a SNR above 40 dB.

6 Conclusion

The encoder proceeds differently that the standard LZW algorithm that always satisfies eq. (3). The encoder in P-LLZW takes different decisions than the standard LZW algorith, because it satisfies eq. (8) rather than eq. (3) as it searches through its dictionary for a match. However, the algorithm is *decode compatible* with the standard LZW algorithm. This means, our implementation is decode compatible with standard GIF readers. Files produced by our implementation are readable by any web browser or any other image processing software that has a GIF loader and complies to the GIF89a standard. This makes algorithm P-LLZW a viable alternative to basic, standard GIF encoders.

					P-LLZW			
8 BPP File	Dimensions	Raw Size	Gif Size	Gif BPP	ϕ	Size	BPP	SNR
"NY Cargo"	1279 × 865	1.1M	766.8K	5.7	10	600K	4.5	34 dB
					3	617K	4.6	47 dB
"Cat"	320 × 200	64000	56.4K	7.2	10	25.2K	3.2	30 dB
					4	49.1K	6.29	44 dB
"100 Francs"	3306 × 1824	5.9M	3.5 M	4.8	10	2M	2.8	31 dB
					3	3.3M	4.6	48 dB
"Jupiter"	3287 × 2475	7.9M	4.5 M	4.6	10	2.2M	2.3	31 dB
					4	3.8M	3.9	43 dB
"Lagaffe"	2550 × 3417	8.5M	5.0M	4.8	10	2.5M	2.4	32 dB
					4	3.9M	3.8	40 dB
"New River"	2428 × 6887	16.7M	7.5M	3.7	10	4.3M	2.15	35 dB
					4	7.1M	3.6	45 dB
"Forest"	1948 × 2966	5.6M	4.1M	5.9	10	2.8M	2.8	29 dB
					4	3.9M	5.7	40 dB
"P-Boxes"	2400 × 3156	7.4M	3.4M	3.7	10	1.1M	1.2	34 dB
					4	2.4M	2.7	43 dB
"Whitman"	1180 × 2192	2.5M	1.5M	4.9	10	993K	3.1	37 dB
					4	1.4M	4.5	55 dB

Table 1: Results with the metric m in eq. (1) and eq. (2) and the objective function described in eq. (10). ϕ ranges from 0 to $255\sqrt{3}$ rather than between 0 and $\sqrt{3}$. $\phi = 10$ implies that the tolerable error is at worst $\approx 3.9\%$ of the dynamic range. The images are all natural images or scanned printed material (at 300 DPI).

One can change the objective function to suit his own needs while remaining compatible with a standard decoder. The algorithm complexity depends on the metric and the objective function. In our implementation, the complexity of search is upper-bounded by the total number of symbols in the dictionary. The overall complexity is then $O(D_{max} \times S)$, where D_{max} is the maximum number of symbols contained by the dictionary (in GIF implementation, it is at most 4096, including the two reserved control symbols) and S is the length of the string to compress.

While the algorithm tries to optimize compression using the objective function, it remains sub-optimal. A truly optimal algorithm would not only search for the best match at time t under some objective function, it would do so considering all possible choices at previous times and at future times. Since the decision took at time t impacts on further decisions, one cannot be sure that any individual decision is optimal unless he considers all possible decisions over all possible times and picks the decision path that optimize globaly compression and image quality, according to his objective function. This, although not impossible to implement, would ask for an unacceptable amout of computation

for maybe not such a great gain. The resulting algorithm would be at least $O(D_{max}^2 \times S)$, even if it is amenable to dynamic programming.

In conclusion, this algorithm, P-LLZW, is a good alternative to basic LZW when losslessness is not necessary. It is better than other simple greedy algorithm that do not optimize for both compression and image quality, such as the two algorithms we presented as prior work. It can also be used with different objective functions to meet different needs. It is also possible to implement the algorithm in such a way that it is compatible with an existing decoder (weither it is image data or some other kind of data). This means that it is not limited to GIF: it could be used in any compression utilty that uses LZW. The difference with GIF and some other data format using LZW compression is likely to be only in the way in which the indexes are encoded in the compressed data stream.

References

[Chiang98] Loben Chiang, *untranslated thesis title*, City University of Hong Kong, August 1998

[Comp89] *The Graphics Interchange Format, v89a*, Compuserve Incorporated, Columbus, Ohio, 1990

[MacAdam42] D.L. MacAdam, *Visual sensitivities to color differences in daylight*, Journal of the Optical Society of America, v32, pp. 247-273, 1942

[Pigeon95] Steven Pigeon, FLATLAND, *ou comment réduire une image GIF en modifiant l'algorithme LZW*, Journal l'Interactif, March 1996

[Welch84] Welch, Terry A., *A technique for high-performance data compression*, Computer, June 1984, pp. 8-19