PRACTICAL IMPLEMENTATION OF THE SPIHT ALGORITHM

Ing. René Puchinger

2013

1 Preface

This text loosely follows my document Concise Introduction to Wavelets and is focused on practical implementation of the Set Partitioning in Hierarchical Trees (SPIHT) algorithm written in C++.

The implementation is as minimal as possible. The compiled program runs as a console application, no GUI is provided.

You can obtain the source code from `https://github.com/renp/imshrinker`

René Puchinger, author.

2 Introduction

Wavelets have found successful applications in signal compression. This text is focused on 2D images.

There are various approaches to this problem. A fairly intuitive method can be developed as follows. Having an image data, we compute the 2D wavelet transform and then threshold the coefficients by means of hard- or soft-thresholding. By rounding the coefficients we achieve them to be represented in fixed number of bits. After these operations we obtain a sparse representation of the original image, i.e., we end up with the same number of different values of which lots of them are zero. We can then compress the thresholded coefficients using an entropy coding method such as Huffman or arithmetic coding. The thresholding and rounding are irreversible operations which implies that the compression is indeed lossy. The amount of distortion is determinded by the threshold value. For a high threshold we obtain more compressible representation at the cost of larger amount of distortion.

3 The SPIHT algorithm

There are two significant drawbacks of the previously mentioned compression method.

1. The amount of distortion depends on the choice of a suitable threshold. The lower the threshold is, the better approximation we obtain, at the cost of larger compressed file size. The ideal threshold value and the corresponding compressed file size are unknown and there is no unique correspondence between these two parameters.

2. Experience shows that there are similarities of wavelet coefficients across individual subbands. For example, regions with values close to zero in the lower subbands tend to be replicated in the higher subbands. The same applies to coefficients corresponding to sharp edges in the original image. The previously described method clearly does not take advantage of this phenomenon.

The SPIHT (Set Partitioning In Hierarchical Trees) algorithm solves these issues by a sophisticated and effective method for transmition of the wavelet coefficents so that for a given bit rate the reconstructed image quality is the best. Moreover, the coefficients are organized in hierarchical tree structures to utilize the similarities across subbands. The algorithm was proposed by A. Said and W. Pearlman [2]. Other good references are [1, 4, 3].

First we will present a simplified version of the algorithm to illustrate the main ideas. For this reason we need some measure of distortion. SPIHT uses the mean squared error

$$D_{MSE} = \frac{1}{N} \sum_i \sum_j (c_{ij} - \hat{c}_{ij})^2,$$

where c_{ij} are the wavelet coefficients of the original image and $\hat{c}_{ij}$ are the wavelet coefficients of the reconstructed image. It turns out that the mean squared error decreases by c_{ij}^2/N when the decoder receives an exact value of the coefficient c_{ij}. This implies that for an efficient transmition the largest coefficients should be transmitted first.

Let us now examine how the SPIHT algorithm utilizes this idea. It sorts the coefficients by their magnitudes (i.e., ignoring the signs) and transmits the highest bits first. As an example, let us consider the 9 coefficients listed in table 1. The coefficient $c_{1,2}$ is $s_1 1 a_4 a_3 a_2 a_1 a_0$, where s_1 is a sign bit and $a_4, \ldots, a_0$ are some bits. The coefficient $c_{0,1}$ is $s_2 1 b_4 b_3 b_2 b_1 b_0$ and so on. We assume that all the coefficients are sorted by their magnitudes so that $|c_{m(k)}| \geq |c_{m(k+1)}|$, where $m : \mathbb{N} \mapsto \mathbb{N}_0 \times \mathbb{N}_0$ is the sorting information that must be transmitted together with the coefficient bits (later we will show a more efficient way to transmit this information).

Table 1: Example

k		1	2	3	4	5	6	7	8	9
	sign	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
msb	5	1	1	0	0	0	0	0	0	0
	4	a_4	b_4	1	1	1	0	0	0	0
	3	a_3	b_3	c_3	d_3	e_3	1	1	1	1
	2	a_2	b_2	c_2	d_2	e_2	f_2	g_2	h_2	i_2
	1	a_1	b_1	c_1	d_1	e_1	f_1	g_1	h_1	i_1
lsb	0	a_0	b_0	c_0	d_0	e_0	f_0	g_0	h_0	i_0
$m(k) = (i, j)$		$(1, 2)$	$(0, 1)$	$(3, 0)$	$(1, 0)$	$(1, 1)$	$(2, 1)$	$(3, 1)$	$(1, 3)$	$(3, 2)$

The encoder performs a number of iterations, where each iteration consists of a **sorting pass** and a **refinement pass**. The refinement pass is not performed in the first iteration. In the first iteration, during the sorting pass, we transmit the number $l = 2$ of coefficients c_{ij} that satisfy $2^4 \leq |c_{ij}| < 2^5$, followed by the signs s_1, s_2 of these coefficients and by the two pairs of coordinates $(1, 2)$ and $(0, 1)$. This finishes the first iteration. The decoder can eventually reconstruct the coefficients $c_{1,2}$ and $c_{0,1}$ as

$$c_{1,2} = s_1 100000,$$

$$c_{0,1} = s_2 100000.$$

In the second iteration, during the sorting pass, we transmit the number $l = 3$ of coefficients that satisfy $2^3 \leq |c_{ij}| < 2^4$, followed by the signs s_3, s_4, s_5 and by the three pairs of coordinates $(3, 0), (1, 0), (1, 1)$. In the refinement pass, we transmit the bits a_4 and b_4. The decoder can then reconstruct the coefficients $c_{3,0}, c_{1,0}, c_{1,1}$ as

$$c_{3,0} = s_3 010000,$$

$$c_{1,0} = s_4 010000,$$

$$c_{1,1} = s_5 010000.$$

The coefficients $c_{1,2}$ and $c_{0,1}$ are then refined as

$$c_{1,2} = s_1 1 a_4 0000,$$

$$c_{0,1} = s_2 1 b_4 0000.$$

In this way the coefficients are successively refined, starting with the most important bits first. We can iterate the process until all the bits are transmitted or we may stop at any point. This is the natural lossy option of the SPIHT algorithm.

The (simplified) algorithm can be summarized as follows:

1. Output $n = \lfloor \log_2(\max_{i,j} |c_{ij}|) \rfloor$, i.e. n initially satisfies $2^n \leq \max_{i,j} |c_{ij}| < 2^{n+1}$.

2. **Sorting pass:** Output l = number of coefficients satisfying $2^n \leq |c_{ij}| < 2^{n+1}$, followed by the coefficient coordinates $m(k)$ and by the sign of each of the l coefficients.

3. **Refinement pass:** Output the n-th most significant bit of all the coefficients satisfying $|c_{ij}| \geq 2^{n+1}$, in the same order as in the sorting pass.

4. If $n = 0$, stop; else set $n := n - 1$ and go to step 2.

The method we have just discussed is fairly simple but it has a considerable drawback – the ordering information must be explicitly transmitted. It turns out that this information can be transmitted indirectly if both the encoder and decoder share the same branching rules. Moreover, there is no need to sort all the coefficients – in each iteration it suffices to select those coefficients that satisfy $2^n \leq |c_{ij}| < 2^{n+1}$. A coefficient c_{ij} satisfying $2^n \leq |c_{ij}|$ is called significant with respect to the threshold 2^n, otherwise it is called insignificant.

This significance test can be extended to sets of coefficients $\mathcal{T}$ so that

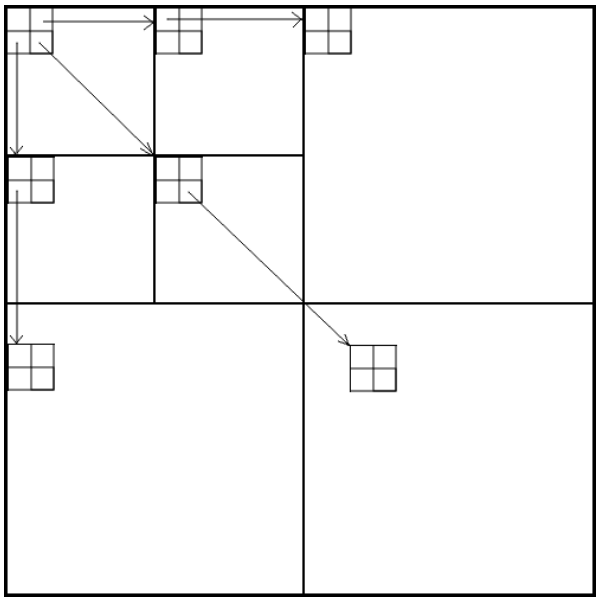
$$S_n(\mathcal{T}) = \begin{cases} 1, & \text{if } \max_{(i,j)\in\mathcal{T}} |c_{ij}| \geq 2^n \\ 0, & \text{otherwise.} \end{cases}$$

In other words, if $S_n(\mathcal{T}) = 0$ then all coefficients in $\mathcal{T}$ are insignificant and if $S_n(\mathcal{T}) = 1$ then some coefficients are significant.

In each sorting pass the partitioning subsets $\mathcal{T}_m$ are tested for significance and if some subset $\mathcal{T}_k$ is found significant then it is partitioned into new subsets $\mathcal{T}_{k,l}$ (more on this later) and the significance test is applied to these new subsets. The set partitioning sorting algorithm creates the partitions in such way that the sets considered significant contain only one element, while the sets considered insignificant contain a large number of elements.

Now that we have roughly described the set partitioning sorting method, we will deal with spatial orientation trees. A spatial orientation tree is a special tree structure which utilizes similarities between the wavelet coefficients across individual subbands (see Figure 1). The tree roots are situated in the highest subbands and are always present in a group of 2×2 coefficients. From these four coefficients the upper-left root

Figure 1: A spatial orientation tree



has no descendants. Except for the highest and lowest levels, the direct descendants (offspring) of the node (i, j) are the nodes $\{(2i, 2j), (2i+1, 2j), (2i, 2j+1), (2i+1, 2j+1)\}$.

Let us denote:

- $\mathcal{H}$: The set of coordinates of all spatial orientation tree roots located in the highest subband.
- $\mathcal{O}(i, j)$: The set of coordinates of the four offspring of the node (i, j). $\mathcal{O}(i, j)$ is empty if (i, j) is a leaf of the tree.
- $\mathcal{D}(i, j)$: The set of coordinates of all descendants of node (i, j).
- $\mathcal{L}(i, j)$: The difference set $\mathcal{D}(i, j) - \mathcal{O}(i, j)$, i.e. the set of coordinates of descendants of the node (i, j), grandchildren onward.

The set partitioning sorting algorithm can now be described by means of spatial orientation trees:

1. The initial sets are $\{(i, j)\}$ and $\mathcal{D}(i, j)$ for all $(i, j) \in \mathcal{H}$.
2. If the set $\mathcal{D}(i, j)$ is significant then it is partitioned into $\mathcal{L}(i, j)$ and the four single-element sets with $(k, l) \in \mathcal{O}(i, j)$.

3. If $\mathcal{L}(i,j)$ is significant then it is partitioned into the four sets $\mathcal{D}(k,l)$, where (k,l) are the offspring of (i,j).

The standard implementation of SPIHT uses three lists - list of insignificant pixels (LIP), list of significant pixels (LSP) and list of insignificant sets (LIS). The LIS contains sets of type $\mathcal{D}(i,j)$ and $\mathcal{L}(i,j)$; they are called type A and type B entries, respectively. In each sorting pass all entries of LIP are tested for significance and if found significant, they are moved to LSP. Similarly, all sets in LIS are tested for significance and if found significant, they are partitioned and each of the new sets is tested for significance, too. In the refinement pass the algorithm transmits the n-th most significant bits of all the entries in LSP, except for those processed in the current sorting pass. The **full SPIHT algorithm** for encoding follows [2].

1. Initialize $n := \lfloor \log_2(\max_{i,j} |c_{ij}|) \rfloor$ and output n. Set the LSP as an empty list and add all coordinates $(i,j) \in \mathcal{H}$ to the LIP. Add all $(i,j) \in \mathcal{H}$ which have descendants to the LIS, as type A entries.

2. Sorting pass:

   2.1 For each entry (i,j) in the LIP do:

      2.1.1 Output $S_n(i,j)$,

      2.1.2 If $S_n(i,j) = 1$ then output the sign of c_{ij} and move (i,j) to the LSP.

   2.2 For each entry (i,j) in the LIS do:

      2.2.1 If the entry is of type A then

      - Output $S_n(\mathcal{D}(i,j))$,
      - If $S_n(\mathcal{D}(i,j)) = 1$ then
        - For each $(k,l) \in \mathcal{O}(i,j)$ do:
          - Output $S_n(k,l)$.
          - If $S_n(k,l) = 1$ then output the sign of c_{kl} and add (k,l) to the LSP.
          - If $S_n(k,l) = 0$ then add (k,l) to the LIP.
        - If $\mathcal{L}(i,j) \neq \emptyset$ then move (i,j) to the end of the LIS, as an entry of type B; else, remove entry (i,j) from the LIS.

      2.2.2 If the entry is of type B then

      - Output $S_n(\mathcal{L}(i,j))$.
      - If $S_n(\mathcal{L}(i,j)) = 1$ then
        - Add each $(k,l) \in \mathcal{O}(i,j)$ to the LIS as an entry of type A.

– Remove (i, j) from the LIS.

3. Refinement pass: for each entry (i, j) in the LSP, except those added in the last sorting pass, output the n-th most most significant bit of $|c_{ij}|$.

4. If $n = 0$, stop; else set $n := n - 1$ and go to step 2.

The decoding algorithm must use the same branching as the encoding algorithm. Therefore, it suffices to replace each word "output" with the word "input". Furthermore, the decoding algorithm must reconstruct the original coefficients. Each time an entry is moved to the LSP, we know that it satisfies $|c_{ij}| \geq 2^n$ so we set $\hat{c}_{ij} = \pm 2^n$, according to whether it is positive or negative. During the refinement pass, all the coefficients in the LSP are updated to the appropriate values, i.e., the n-th bits are set either to 0 or 1, depending on the input.

We will demonstrate few iterations of the full SPIHT algorithm on an example (see Table 2). We initialize $n := \lfloor \log_2 30 \rfloor = 4$ and transmit n. The lists are set as follows:

Table 2: SPIHT coding example

30	10	8	5
12	-9	5	-6
-7	3	2	-1
5	2	1	0

$$\text{LIP} := \{(0,0), (0,1), (1,0), (1,1)\},$$

$$\text{LIS} := \{(0,1,A), (1,0,A), (1,1,A)\},$$

$$\text{LSP} := \{\}.$$

1. We have $n = 4$, threshold $T = 16$. In the sorting pass we first process LIP: $c_{0,0}$ is significant and positive, so we transmit 10 and move $(0, 0)$ to LSP; $c_{0,1}, c_{1,0}$ and $c_{1,1}$ are insignificant so we transmit a 0 for each. Now we process LIS: all the sets in LIS are currently insignificant so we transmit three 0. The refinement pass is not performed in the first iteration. We have thus transmitted 10000000. The lists are now

$$\text{LIP} := \{(0,1), (1,0), (1,1)\},$$

$$\text{LIS} := \{(0,1,A), (1,0,A), (1,1,A)\},$$

$$\text{LSP} := \{(0,0)\}.$$

2. Now $n = 3$, $T = 8$. Sorting pass. Process LIP: $c_{0,1}$ is significant and positive, move it to the LSP and transmit 10; the same for $c_{0,1}$; $c_{1,1}$ is significant and negative, move it to the LSP and transmit 11. Process LIS: the set $(0, 1, A)$ is significant, transmit 1 and process the four offspring of $(0, 1)$: $c_{0,2}$ is significant and positive, transmit 10 and add $(0, 2)$ to the LSP. $c_{0,3}, c_{1,2}$ and $c_{1,3}$ are insignificant so transmit a 0 for each and add them to the LIP. The sets $(1, 0, A)$ and $(1, 1, A)$ are insignificant so transmit a 0 for each. Since $\mathcal{L}(0, 1) = \emptyset$, remove the entry $(0, 1, A)$ from the LIS. Refinement pass: transmit the third bit of $c_{0,0} = 30$, which is 1. In total we have transmitted 101011110000001. The state of the lists is

$$\text{LIP} := \{(0, 3), (1, 2), (1, 3)\},$$

$$\text{LIS} := \{(1, 0, A), (1, 1, A)\},$$

$$\text{LSP} := \{(0, 0), (0, 1), (1, 0), (1, 1), (0, 2)\}.$$

3. Now $n = 2$, $T = 4$. Sorting pass. Process LIP: move $(0, 3), (1, 2)$ and $(1, 3)$ to the LSP and transmit 10, 10, and 11. Process LIS: The set $(1, 0, A)$ is significant so transmit a 1 and process its four offspring: $(2, 0) \to$ LSP, transmit 11; $(2, 1) \to$ LIP, transmit 0; $(3, 0) \to$ LSP, transmit 10; $(3, 1) \to$ LIP, transmit 0. Since $\mathcal{L}(1, 0) = \emptyset$, remove the entry $(1, 0, A)$ from the LIS. The set $(1, 1, A)$ is insignificant so transmit a 0. Refinement pass: transmit the second bits of $c_{0,0}, c_{0,1}, c_{1,0}, c_{1,1}, c_{0,2}$, which are 1, 0, 1, 0, 0. In total we have transmitted 1010111110100010100. The lists are now

$$\text{LIP} := \{(2, 1), (3, 1)\},$$

$$\text{LIS} := \{(1, 1, A)\},$$

$$\text{LSP} := \{(0, 0), (0, 1), (1, 0), (1, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (3, 0)\}.$$

We decide to stop at this point (of course we could have stopped at *any* point during the process).

The decoding algorithm follows the same branching path as the encoding algorithm. We only summarize the states of the wavelet coefficients in each step - see Tables 3 - 6.

Table 3: SPIHT decoding example - state after initialization

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Table 4: SPIHT decoding example - state after the first step

16	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Table 5: SPIHT decoding example - state after the second step

24	8	8	0
8	-8	0	0
0	0	0	0
0	0	0	0

Table 6: SPIHT decoding example - state after the third step

28	8	8	4
12	-8	4	-4
-4	0	0	0
4	0	0	0

4 Comparison of encoders

Herein we compare the results of image compression using the ImShrinker and the popular JPEG, which incorporates the discrete cosine transform, applied to blocks of 8 $\times$ 8 pixels.

The original image is in public domain and the dimensions are 512 $\times$ 512 pixels.

From the figures it follows that ImShrinker performs better at extra low bit-rates, while at common bit-rates (1.5 bps or more) the differences between JPEG and ImShrinker are comparable. It also follows that the artifacts in JPEG are blocky while in ImShrinker the images are smoothed.

One particular disadvantage of ImShrinker are additional artifacts near the edges, which are caused by the fact that the wavelets are not adapted near the edges. There exists a modifcation in the lifting scheme which adapts the wavelets so that the distortion is not so visible. It has not been implemented, though.

Also note that an algorithm similar to SPIHT has been implemented in the newer JPEG2000 standard.

Figure 2: Left: JPEG (37364 bytes). Right: ImShrinker - SPIHT (37201 bytes).

Figure 3: Left: JPEG (3618 bytes). Right: ImShrinker - SPIHT (3584 bytes).

A Code overview

Herein we summarize the C++ classes from which the application is composed. Full source code can be found at `https://github.com/renp/imshrinker`.

1. `class Application` - main application class,
2. `class BitInputStream` - class for bit oriented I/O handling,
3. `class BitOutputStream` - class for bit oriented I/O handling,
4. `class Decoder` - class for wavelet image decompression,
5. `class Encoder` - class for wavelet image compression,
6. `class Exception` - class for error handling,
7. `class FileInputStream` - class for buffered I/O handling,
8. `class FileOutputStream` - class for buffered I/O handling,
9. `class Image` - class for PGM/PPM file image reading and writing,
10. `class IOStream` - stream interface,
11. `class SPIHT_Encoder` - class for wavelet coefficients coding using the SPIHT algorithm,

12. `class SPIHT_Decoder` - class for wavelet coefficients decoding using the SPIHT algorithm.

One can compile the program in Linux or by means of eg. Cygwin using the command `g++ *.cc -o imshrinker`.

References

[1] R. M. Rao and A. S. Bopardikar, *Wavelet Transforms - Introduction to Theory and Applications*, Addison Wesley, 1998.

[2] A. Said and W. A. Pearlman, *A new, fast, and efficient image codec based on set partitioning in hierarchical trees*, IEEE Transactions on Circuits and Systems for Video Technology, 6 (1996), pp. 243–250.

[3] D. Salomon, *Data Compression - The Complete Reference*, Springer Verlag, third ed., 2004.

[4] K. Sayood, *Introduction to Data Compression*, Morgan Kaufmann, second ed., 2000.