

Hierarchical Context-based Pixel Ordering

Ziv Bar-Joseph* and Daniel Cohen-Or†

* MIT Lab for Computer Science

†School of Computer Science, Tel-Aviv University

Abstract

We present a context-based scanning algorithm which reorders the input image using a hierarchical representation of the image. Our algorithm optimally orders (permutes) the leaves corresponding to the pixels, by minimizing the sum of distances between neighboring pixels. The reordering results in an improved autocorrelation between nearby pixels which leads to a smoother image. This allows us, for the first time, to improve image compression rates using context-based scans. The results presented in this paper greatly improve upon previous work in both compression rate and running time.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling I.3.6 [Computer Graphics]: Methodology and Techniques

Keywords: image-space, context-based, lossless compression, leaf ordering, quadrees.

1. Introduction

Many image-space algorithms work on a *linearized* version of the 2D input image, assuming a strong coherence among nearby pixels. The most common way is a scan-line order, where the pixels are traversed horizontally line by line. However, the spatial coherence among the nearby pixels is typically anisotropic, rather than directional, along the horizontal scan lines. Other scan methods can better retain the coherence in the linearized sequence, by taking advantage of the *local* similarities inherent in images⁹. One popular such scan is the Peano-Hilbert space filling curve. This scan recursively traverses each quadrant entirely before moving to the next quadrant (see Figure 1), and thus increases the pixel similarities among neighboring pixels in the scan.

The sequence of pixels visited along the scan defines a one-to-one mapping of the image pixels into a linear sequence. We call this mapping a *pixel ordering*, and it can be regarded as a permutation of the original scan-line order. As the ordering goal is to increase the autocorrelation of the resulting pixel sequence, a good pixel ordering is one that minimizes some metric among the pixels. Typically one would like to minimize the distances among adjacent pixels in the sequence.

This work was geared to compute effective pixel-orderings for lossless image compression. For example, one of the most popular compression formats is GIF (Graphics Interchange Format), which is based on the Lempel-Ziv (LZ) sequence compression algorithm⁷. This and other lossless encoding methods are applied over a sequence of values and thus requires the transformation of the image into a linear sequence of pixels. Specifically, it has been shown that when compared to line scan, the Peano-Hilbert scan improves the compression rates, both theoretically⁸ and in practice^{10,1}

The Peano-Hilbert scan, like other similar space-filling curves, is a universal one in the sense that it is defined independently of the context of the underlying image it traverses. Recently, Dafner *et al*⁴ introduced a *context-based* pixel-ordering algorithm. Unlike previous orderings, their method uses the specific context of the input image to determine the resulting sequence order. Their algorithm constructs a spanning tree based on the input image in $O(n \log^* n)$ time, and uses this spanning tree to reorder the input image. They show that a context-based approach improves the autocorrelation of the new pixel ordering. To retrieve the original image, a context-based pixel ordering necessarily needs to associate the specific ordering with the given image. Encoding the associated ordering imposes a space overhead. In their paper,

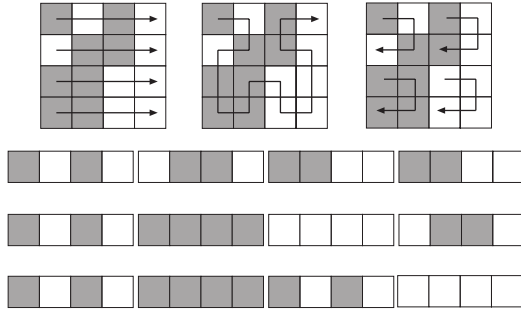


Figure 1: Scanning methods: (a) Line scan (b) Peano-Hilbert scan (c) A recursive quadtree scan, where each level is scanned clockwise from top-left, right, down to bottom-left. Note that (a) and (c) connect pixels that are not adjacent in the image.

Dafner *et al.*⁴ show that the encoding cost hinders the effectiveness of the context-based ordering for lossless compression. While the re-ordered image compresses 5-10% better than the input image, when taking into account the penalty of encoding the spanning tree, the resulting size is 4-7% larger when compared with the compressed original image.

It should be noted that the method of Dafner *et al.*⁴ defines a scan order that is a space-filling-curve. That is, two consecutive pixels in the linear order are adjacent neighbors in the original image. This is not a necessary requirement since any permutation of the pixels is a valid pixel ordering. However, a naive encoding of an arbitrary permutation is overly expensive. Thus, the challenge is to define a sub-optimal effective permutation that has a compact encoding for its associated ordering.

In this paper we introduce a novel approach for pixel ordering. Instead of directly ordering the pixels of the input image, our algorithm works on a hierarchical representation of the image. While hierarchical image representation was used in many graphics applications in the past^{5,11}, to the best of our knowledge, this is the first time such representation is used to improve context based image scans. We begin by generating a full quadtree from the input image, where internal nodes are the average of their four sub-pixels. Next, we reorder the nodes of the tree such that the distances between the leaves, representing the pixels in the resulting tree, are minimized. To achieve a compact encoding of the ordering, we use the ordering of upper levels in the tree to *learn* an ordering for lower levels. Since most of the internal nodes reside in the lower levels of the tree, this step results in a large reduction in the size of the encoding. Specifically, our context-based pixel ordering technique is shown to be effective for lossless image compression even when the cost of encoding the ordering is included.

The major conceptual difference between our ordering method and all previous methods is that our pixel order-

ing takes advantage of *global* similarities. Our algorithm can connect remote regions in the image which are similar. This results in a longer similar sequence, with better auto-correlation, which significantly improves compression (see Figure 2). It should be noted that finding similar regions in the image is used to improve *lossy* image compression⁶, however, to the best of our knowledge this is the first time it has been successfully used for lossless image compression. Our algorithm improves the compression rates by up to 9 percentage points when compared to the line scan result, and up to 4 percentage points compared to the Peano scan result. Since lossless compression rates are usually between 20-40%, this is a significant improvement.

The rest of the paper is organized as follows. The next section presents an overview of our ordering algorithm. In Section 3 we formulate the optimal ordering problem, and in Section 4 we discuss our algorithm in details. In Section 5 we discuss the complexity of our algorithm and in Section 6 we discuss a number of optimization methods that are used to reduce the mapping size while still achieving a good ordering. In Section 7 we compare the compression results achieved by our algorithm to those achieved using other scan techniques. Section 8 concludes this paper and discusses directions for future work.

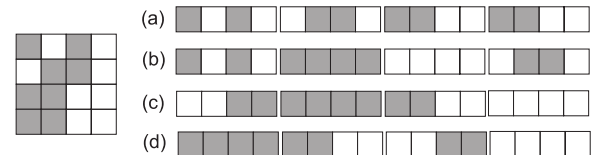


Figure 2: Four scans of the little image on the left. (a) Line scan (b) Peano-Hilbert scan (c) Optimal leaf ordering (d) level ordering. As can be seen, both context-based scans ((c) and (d)) have much fewer transitions from gray to white and vice versa (2 and 3, respectively), and thus tends to compress much better.

2. Hierarchical Pixel ordering

Assume an $N \times N$ image where $N = 2^k$. We first construct a full quadtree from the input image in the following way. Starting with the input image we associate each pixel value with a leaf (terminal node) in the tree. Recursively, each internal node is assigned with the average values of its four sons. Fixing an order among the four sons of each internal node results in a linear sequence of image pixels, defined by the standard in-order traversal of the tree. The pixel ordering problem is to reorder the leaves of this quadtree aiming to minimize the sum of the distances among adjacent leaves (pixels) in the linear order.

The problem of optimally reordering the leaves of a tree in order to reduce the sum of their distances was first introduced in the context of computational biology³. In that paper the authors improve hierarchical clustering by reordering

the leaves of the clustering tree. While this algorithm can be used to reorder small input images, it is not suitable for common and large images due to its computational complexity (see Section 3). Thus, in this paper we introduce an algorithm which solves a restricted version of the optimal ordering problem. We look for the optimal order of level l , given that the nodes above that level (i.e. the nodes between level l and the root) are fixed. For a level with n nodes, the number of possible orderings is $24^{n/4}$, since there are $4! = 24$ possible orderings for each node, and $n/4$ nodes in level $l - 1$. We present an algorithm that, given an input image, solves the ordering problem in $O(n)$ time and $O(n)$ space, and thus is feasible for arbitrary large images.

We first present a high level description of our algorithm. In Section 4 we discuss the algorithm in more detail, and prove that its running time and space complexity is $O(n)$.

The pixel ordering is achieved by ordering the full quadtree, based on a top-down ordering of each level separately. Figure 3 illustrates the tree. First, level 1 is ordered by trying all 24 possible permutations. Now, assume we have ordered all levels up to level k , and we have to order level k , keeping all the nodes above level $k - 1$ fixed.

Ordering level k consists of a series of re-orderings of the sons of the nodes on level $k - 1$. Though the orderings are local, the way we find the best order is by optimizing a global function which searches all $24^{n/4}$ possible orderings for the one that minimizes the sum of distances.

From now on we refer to the tree presented in Figure 3. Any global order will have a leaf from LS1 as its leftmost leaf, and a leaf from RS4 as its rightmost leaf. Thus, it suffices to find the best global order for any pair of these leaves, and choose the pair that minimizes the sum of distances of neighboring leaves in the ordering. This is done recursively. Assume we have already computed the best subtree order for every pair of rightmost and leftmost leaves for the subtrees rooted at S1, S2, S3 and S4. Denote this quantity, for the rightmost and leftmost pair of leaves l_1 and r_2 in subtree S1, by $M^{S1}(l_1, r_2)$. We first combine S3 and S4 in the following way.

When combining S3 and S4, the leaves that connect the two subtrees come from RS3 and LS4, while the rightmost and leftmost leaves of the combined subtree, S34, are from LS3 and RS4. Since we have already computed the best ordering for every pair of leaves from LS3 and RS3, and for every pair from LS4 and RS4, all that is left is to go over all pairs of leaves from LS3 and RS4. For each such pair $l_1 \in LS3$ and $r_4 \in RS4$, we go over all possible pairs from $r_2 \in RS3$ and $l_3 \in LS4$, and compute the following quantity:

$$C(r_2, l_3) = M^{S3}(l_1, r_2) + d(r_2, l_3) + M^{S4}(l_3, r_4)$$

where $d(\cdot)$ is the distance metric. We set $M^{S34}(l_1, r_4)$ to be $C(r_2, l_3)$ for the pair r_2, l_3 that minimizes C for l_1

and r_4 . After going over all possible pairs, we end up with $M^{S34}(l_1, r_4)$ for all pairs of leftmost and rightmost leaves.

Now we repeat the same process with S1 and S2, resulting in $M^{S12}(\cdot, \cdot)$. The final step is to combine S12 with S34, which is done in the same way we combined S3 and S4, though this time we use $M^{S12}(\cdot, \cdot)$ and $M^{S34}(\cdot, \cdot)$. After this step, we choose the pair of leaves from S1 and S4 that minimizes $M^R(\cdot, \cdot)$, we backtrack to see how we reached this value, and reorder the leaves of the tree accordingly.

3. Optimal Leaf Ordering

Optimal leaf ordering is the problem of finding the ordering that minimizes the sum of the differences between adjacent leaves of a given tree. This can be done by flipping internal nodes until such a minimum is obtained. More formally, for a tree T with n leaves (where $n = 4^l$), denote by v_1, \dots, v_n the leaves of T and by $x_1 \dots x_{n-1}$ the internal nodes of T . A **linear ordering consistent with T** is defined to be an ordering of the leaves of T generated by flipping internal nodes in T (that is, changing the order between the four subtrees rooted at x_i , for any $x_i \in T$). In our case there are $(n - 1)/3$ internal nodes in the tree, and thus there are $24^{(n-1)/3}$ possible orderings of the leaves of the tree. Denote by Φ the space of all the possible orderings of the tree leaves. For $\phi \in \Phi$ we define $D^\phi(T)$ to be:

$$D^\phi(T) = \sum_{i=1}^{n-1} S(v_{\phi_i}, v_{\phi_{i+1}})$$

where $S(v, w)$ is any distance measure between two leaves of the tree (we will discuss the actual distance metric we use in Section 7). We define $D(T)$ as:

$$D(T) = \min_{\phi \in \Phi} (D^\phi(T))$$

that is, $D(T)$ minimizes the distance between adjacent leaves in the linear ordering, over all possible consistent arrangements of the leaves of T . Our goal is to compute $D(T)$ and find the leaf ordering ϕ for which $D^\phi(T) = D(T)$.

In ² the authors present an algorithm that uses dynamic programming to compute an optimally ordered binary tree in $O(n^3)$ time and $O(n^2)$ space. This algorithm can be extended to quadtrees, though the running time and memory requirements are slightly increased.

The above algorithm is overly expensive when it comes to ordering images. Unlike clustering trees, the number of pixels (or leaves) in an image tree is very large, making an $O(n^3)$ algorithm impractical. Further, the $O(n^2)$ space requirement is also a problem due to the large number of leaves. For 512 X 512 images, even if the algorithm uses only 1 byte for each pair of leaves, the memory requirements will be over 65 Giga bytes. Note that in ² the authors present a lower bound of $O(n^2)$ time for solving the optimal leaf-ordering problem. Thus, even if a new algorithm for this

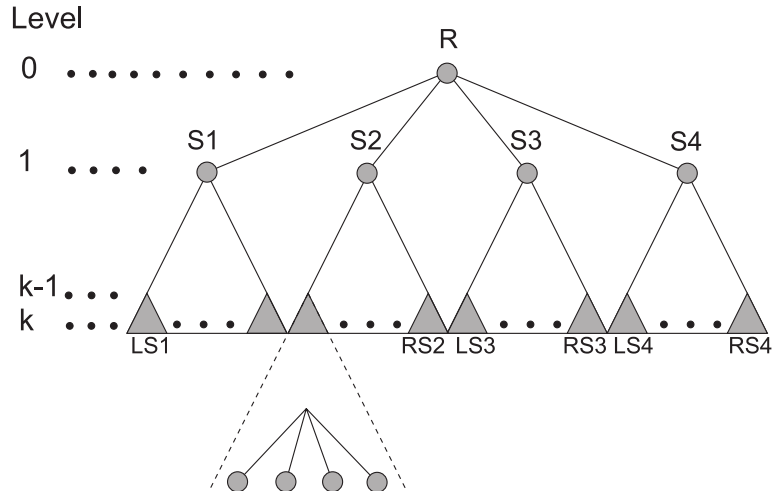


Figure 3: The full quadtree is ordered by an optimal ordering of each level. The little gray triangles in the bottom represent a quad of four leaves. See text for details.

problem is suggested, it will still not be suitable for large images.

4. Ordering the Pixels

In light of the above problems, we have developed an algorithm that solves a restricted version of the optimal leaf-ordering problem. This problem assumes that a distance can be computed not only between different leaves (pixels) of the trees, but also between different internal nodes on the same level in the tree. For such a tree, the *pixel-ordering* problem seeks the optimal ordering of each level, based on the assumption that all levels above this level are fixed.

Note that the number of possible orderings is still exponential in the number of internal nodes (or leaves) at that level. In particular, the number of possible orderings at the lowest level (in which the leaves reside) is $24^{n/4}$, which is still exponential in n . Nonetheless, unlike the optimal ordering problem, the pixel-ordering problem has a solution in time $O(n)$ and space $O(n)$ as we discuss below.

For a level k , the algorithm begins by calling $levelOrdering(v, 1, k, S, lr)$ (see Figures 5 and 6), where v is the root of the tree, S is the distance matrix for level k , and $lr = left$. Since the algorithm is defined by the level k , we will treat the nodes of level k as leaves from now on. The algorithm associates each internal node v with a structure M^v which has three fields: *sources*, *targets* and *dist*. For an internal node v , $M^v.sources$ holds either the leftmost four leaves of v or the rightmost four leaves (depending on the value of the lr parameter). Note that since all levels higher than $k-1$ are fixed, these sets do not change throughout the algorithm (though their internal order can change). $M^v.targets$

holds the complementary set of leaves (the rightmost or leftmost). Assume from now that $M^v.sources$ holds the leftmost four leaves of v (the case of rightmost leaves is symmetric). $M^v.dist(i, j)$ holds the optimal sum of distances between the leaves of v when i is the leftmost leaf and j is the rightmost leaf of v , for all pairs of leaves $i \in M^v.sources$ and $j \in M^v.targets$. If v is the root of the tree, then finding the pair i, j that minimizes $M^v(i, j)$ solves the level-ordering problem, since there must be a leaf from $M^v.sources$ on the leftmost side of v and a leaf from $M^v.targets$ on the rightmost side. Thus, the level-ordering problem reduces to the problem of computing M^v for the root of the tree v .

This is done in the following way. We go down the tree recursively, and at each internal node, compute M^v . If v is a node on level $k-1$, then M^v can be computed by going over all possible orderings of the four leaves that are sons of v . Otherwise, v is an internal node. Assume we have computed M^v for all four of v 's children's. To compute M^v we need to combine these results. First, we combine the two leftmost of v 's children. ($v.sons[1]$ and $v.sons[2]$) in M_{12} . Next we combine the two rightmost of v 's children in M_{34} . Finally, we use M_{12} and M_{34} to compute M^v , according to the lr variable (i.e. - $M^v.sources$ is either the leftmost four leaves or the rightmost depending on lr). This computation is presented in Figure 4. After computing M^v for the root v and finding the pair i, j that minimizes $M^v.dist(i, j)$, we use backtracking to find the path we took to arrive at M^v . This gives the actual ordering of the leaves of T . This can be done in a similar way to the backtracking used in dynamic programming.

The complete algorithm involves l successive calls to $levelOrdering$, starting with $k=0$ (the root) and ending with $k=l-1$. After the last call the resulting tree solves the level-

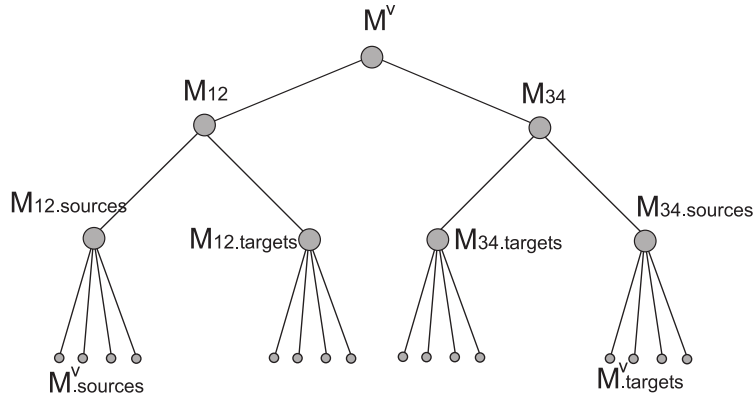


Figure 4: The M^v data structure: Following the computation of M_{12} and M_{34} , we can compute the M^v data structure. In this figure the lr variable is set to 'left' and so the sources for M^v are the leftmost nodes, and the targets are the rightmost nodes. By looping over the leaves in the intersection ($M_{12}.targets$ and $M_{34}.targets$) we compute for every pair of rightmost and leftmost leaves of M^v the level ordering solution for this pair (see also the algorithm in Figures 5 and 6).

ordering problem, as each level is optimally ordered with respect to the levels above it.

5. Algorithm Complexity

In this section we analyze the complexity of the entire algorithm, i.e. the combined complexity of the l calls to *levelOrdering*. Recall that n is the number of leaves (pixels) in the tree, and $l + 1$ is the number of levels in the tree. Thus $4^l = n$. For level k , there are 4^k leaves in the tree that *levelOrdering*(v, k, s, lr) orders. Here we show that the running time of the *levelOrdering* for level k is $O(4^k)$ and the space complexity is also $O(4^k)$. Combining all the runs of *levelOrdering* results in an algorithm with time and space complexity of $O(n)$, as we show below.

For level k , our algorithm goes over all internal nodes on levels $0 \dots k - 1$. For each internal node on level $k - 1$, our algorithm goes over all possible pairs of rightmost and leftmost leaves for this node. There are $4 \times 3/2 = 6$ such pairs, and each requires two summations (since there are two possible orderings of the internal leaves). Thus, the total number of operations for this function is a small constant. For each node on levels $0 \dots k - 2$ the algorithm calls the function *combinePair* three times. Each call to this function results in a constant work (since we need to examine at most 16 leaves). Thus, the total running time of the algorithm for level k is $O(4^k/4 + 4^k/12) = O(4^k)$. For $k = l$, we have $4^k = n$ and thus the total running time of this algorithm is $(n + n/4 + n/16 + \dots + 1) = O(n)$.

As for space complexity, note that once the the algorithm for level k terminates, and the tree is reordered, we do not need to keep the space that was allocated for this call. Thus, we only need to analyze the most expensive call to

levelOrdering which is done when $k = l - 1$. For $k = l - 1$ we have $n/4$ nodes on level $k - 1$, and $n/12$ nodes on levels $0 \dots k - 2$. For each node on level $k - 1$ we need to generate the structure M . Each such structure requires a small constant space. For each node on level $0 \dots k - 2$ we use three M structures. Thus, the total memory requirement is $O(n/4 + n/12) = O(n)$.

While the above analysis discusses the encoding of the image, the decoding step is much faster. Since the reordering is kept with the compressed image, it only requires one pass over the tree to reorder each internal node. Also, since the number of internal nodes is $n/3$ the total time is very close to the time it takes to go over all pixels in the image, which is done regardless of the scanning method used.

Thus, our algorithm is efficient in both memory and space, and is suitable for handling large images with a linear dependency on the image size.

6. Encoding Optimizations

To make context-based pixel ordering useful for image compression, we need to optimize the associated encoding of the mapping. As described in Section 4, for each internal node our algorithm records the mapping it uses. There are $4! = 24$ possible orderings for the sons of each node; thus, each node can be encoded with five bits. Since there are $(n - 1)/3$ internal nodes, the total number of bits required for the entire image is $5n/3$ bits, or $5/3$ bits per pixel. In many cases, the size of the encoded mapping is too large, and can erase the reduction in compressed image size. In this section we present a number of methods which reduce the size of the mapping while minimizing the reduction in the compression

```

levelOrdering(v, level, k, S, lr) {
  If level = k - 1 {
    // v is the immediate parent of the current level
    Mv.sources = Mv.targets = v.sons
    forall i ∈ Mv.sources {
      Mv.dist(i, i) = ∞
      for all j ∈ Mv.targets, j ≠ i {
        {m, n} = v.sons \ {i, j}
        Let min = min{S(i, m) + S(n, j), S(i, n) + S(m, j)}
        Mv.dist(i, j) = S(m, n) + min
      }
    }
    return Mv
  }
  else {
    // we need to combine four internal nodes
    for i = 1 : 4
      M[i] = levelOrdering(v.sons[i], level + 1, S, LR)
      // LR is 'left' for 1,3 and 'right' for 2,4
    M12 = combinePair(M[1], M[2])
    M34 = combinePair(M[4], M[3])
    if lr == left {
      Mv = combinePair(M12, M34)
    } else {
      Mv = combinePair(M34, M12)
    }
    return Mv
  }
}

```

Figure 5: Level Ordering algorithm for level k

rate of the reordered image. In the remainder of this section we assume that the image is of dimensions $2^l \times 2^l$.

6.1. Suboptimal orderings

To decrease the size of the encoded mapping it is possible to reduce the number of possible orderings for each node from $4!$ (24) to 16. After examining a large set of images, we found that in most cases the ordering that results in a diagonal jump between the first and second pixel in the ordering is rarely used. Thus, we can eliminate this move, and allow only clockwise or Counter-clockwise moves from the first pixel in the ordering.

The above constraint reduces the number of possible orderings from 24 to 16 ($4 \times 2 \times 2$), since once the first son is chosen, there are only two possible sons that can follow it. Thus, the encoded mapping size is reduced to $4n/3$.

6.2. Statistical encoding

So far we have assumed that the *levelOrdering* algorithm is applied on all levels: $0 \dots l - 1$. Thus, we need to record the

```

combinePair(M1, M2) {
  Mv.sources = M1.sources
  Mv.targets = M2.sources
  for all i ∈ M1.sources
    for all p ∈ M2.targets
      T(i, p) = minm ∈ M1.targets M1.dist(i, m) + S(m, p)
    }
  for all i ∈ M1.sources
    for all j ∈ M2.sources
      Mv.dist(i, j) = minp ∈ M2.targets T(i, p) + M2.dist(j, p)
    }
  return Mv
}

```

Figure 6: Combining the results from two internal nodes

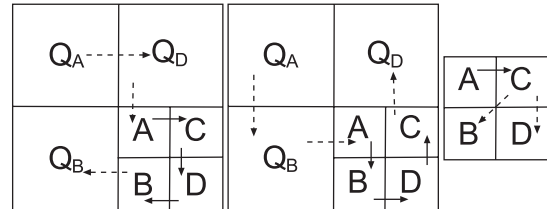


Figure 7: (left) and (middle) Two possible orderings of the sons of Q may result in two different orderings of the sons of Q_C (expanded square on the bottom right side). This indicates that if we do not want to encode level $l - 1$, then by taking into account the position of Q_C in the ordering, as well as its predecessor and successor, we can achieve an improved ordering when compared a pre-specified ordering for all nodes in level $l - 1$. (right) Two of the four cases associated with the second node in the ordering. In this case D is the second node in the ordering, and moving from D 's predecessor to D is done in a clock-wise manner. The two cases differ in the manner we move from D to its successor in the ordering (clock-wise or diagonal). The other two cases for the second node in the ordering arise when the move from D 's predecessor to D is counter clockwise.

ordering of all $n/3$ internal nodes in the image tree. However, $3/4$ of these internal nodes are on level $l - 1$. For large images, the close neighborhood of a pixel is in many cases very similar to the pixel itself. Thus, while ordering higher levels allows us to bring similar remote areas in the image into close proximity, ordering the last level is less beneficial. One way to reduce the size of the mapping is to order only the first $l - 2$ levels, leaving the sons of the nodes on level $l - 1$ in some pre-specified order. This way we only need $4n/12 = n/3$ bits for the mapping.

Though such a pre-specified ordering is a simple solution,

we can do better by using *statistical encoding*. The idea is to encode the ordering associated with a node P on level $l - 1$ as a function of its relative location with respect to its sibling nodes, in the order of its parent Q (which resides in level $l - 2$). For example, suppose that the ordering we computed for the sons of Q is $Q_A \rightarrow Q_D \rightarrow P \rightarrow Q_B$ (where $P = Q_C$, see Figure 7 (a)). Then we can specify the encoding of the sons of P as a function of the ordering of the sons of Q in the following way. First, we use the location of P in the specific ordering of the sons of Q (first, second, etc). A second input to our function is the direction in which we move when going from P 's predecessor to P (clockwise, counter-clockwise or diagonal, see Figure 7(c)). The last input is the direction of the move from P to its successor in the ordering of the sons of Q . Since the number of possible orderings and internal locations of a node in its parent ordering is small, this function can be encoded as a small look-up table.

It turns out that there are a limited number of cases resulting from the above function. More specifically, a look-up table for this function requires only 14 entries (two entries for each of the first and last nodes in the ordering, and 4 for the second and third nodes). Since there are 16 possible permutations or orderings, each entry in the table contains a code of four bits. In total, the size of the look-up table is 14×4 bits only, which is stored only once in the header of the encoded image.

The content of the look-up table is computed by looking at the specific statistics of the input image. This is done as follows. When computing the ordering for level $l - 1$ of the image, we build a histogram by collecting the frequencies of the permutations for each of the 14 possible cases. Each entry in the look-up table encodes the most popular permutation for the case it represents. Next, we reorder the sons of the nodes on level $l - 1$ based on the Look-up table. Note that in order to decode level $l - 1$ the decoder just needs the look-up table (which is stored) and the order of level $l - 2$, which has already been decoded, and so the lossless decoding is guaranteed.

The above statistical encoding of level l does not increase the running time of the algorithm, since we are computing the optimal ordering for level l in any case, and finding the most popular ordering for all cases only takes an additional $n/4$ time (we only need to go over all internal nodes on level $l - 1$).

The above discussion regarding the ordering of level l based only on the ordering of level $l - 1$ can be extended to reduce the levels we order further. For example, if we reduce the levels we encode by two, the encoded mapping size is reduced by 94%. We can repeat the process above by ordering the nodes of both levels (l and $l - 1$) based on the ordering of levels $l - 2$ and $l - 1$. In addition, we can increase the number of possible cases by making the encoding a function of the ordering of the upper two levels rather than just the immediate one. Specifically, when using two levels,

we have $14 \times 14 = 196$ possible cases, which can be stored in a look-up table using only 98 bytes (since it needs 1/2 byte for each entry).

7. Results

In order to test our algorithm, we have used an image compression algorithm to compare our results with previous scanning methods. We have chosen the GIF image compression algorithm, since it is was used in the past to compare both context-less scans^{1,8} and context-based scans⁴, and can thus serve as a good comparison platform for our pixel ordering algorithm. In addition, since GIF is one of the most popular lossless image compression algorithms, improvements to GIF will be of practical use.

When computing an ordering for an input image we can either re-order each of the color channels on its own, or compute one re-ordering for the entire image. While computing separate orderings for each color channel results in a better compression of the re-ordered image, the overhead imposed by such an ordering hinders the effectiveness of such method. Instead, we compute one ordering for the entire image using the following metric for RGB color images:

$$S(i, j) = |i_{red} - j_{red}| + |i_{green} - j_{green}| + |i_{blue} - j_{blue}|$$

As mentioned in the previous section, there are two costs associated with our image scans. The first is the compressed image size and the second is the mapping size. When using the line scan or the Peano scan, the mapping size is 0. For our algorithm the mapping size depends on the image size as well as on the number of levels that are actually ordered (as discussed in Section 6.2). The total compression size is the combined size of the compressed image and the mapping.

In Figure 8 we present three different scans for an example image. As can be seen, using our ordering algorithm results in a scan that is much smoother than the Peano scan. For example, several of the separate white regions in the Peano image are combined in the result generated by our algorithm, and the same holds for the black pixels. Note that this is not surprising as the Peano scan is just one possible outcome of our algorithm.

In Table 7 we present a comparison of the results obtained using GIF on the images in Figures 8 and 10. We compared the results obtained by our algorithm to the results obtained by the line scan (i.e. the original image), and those obtained by the Peano scan. For our algorithm we used two possible image orderings: ordering the entire image tree (as described in Section 4), and ordering all levels except for levels l and $l - 1$ (as discussed in Section 6.2).

In all cases we looked at, compressing the image obtained by using the original *levelOrdering* algorithm resulted in a 10-20 percentage point reduction when compared to both the line scan and the Peano scan. However, when taking into

Figure	Original	Scan line	Peano	Pixel ordering		Optimized		improve. line(%)	improve. Peano(%)
				comp.	total	comp.	total		
room	65536	53694	54412	47378	58300	52957	53739	0	1.2
flowers	262144	189793	183334	153452	197142	175608	178438	6	2.7
texture	262144	170057	170467	145750	189440	161257	164087	3.5	3.7
cells	262144	184396	188080	156755	200445	178695	181515	1.6	3.5
people	1048576	746450	697433	590861	765621	666433	677455	9.2	2.9
lunch	1048576	598552	571860	486095	660855	554198	565220	5.6	1.2

Table 1: Comparison between four possible orderings of the input image when using the GIF compression format. All images were 8 bits/pixel color images. The first column lists the original image size (in bytes). The second and third list the size of the compressed image when line scan and Peano scan are used (respectively). For our algorithm we list two results. The first is the original pixel ordering algorithm from Section 4 and the second is the algorithm in which only levels up to $l-2$ are ordered, as discussed in Section 6.2. For each of these algorithms we present the size of the (compressed) resulting image, and the total size, which is the size of the compressed image plus the mapping size. The last two columns contain the improvement of the Optimized result when compared to the line scan and Peano results. In each row we have highlighted the smallest compression size and the smallest total size. As can be seen, in all cases the best compression was achieved by our original pixel ordering algorithm. However, when taking into account the mapping size the pixel ordering result moves from first to last place, and the best result is obtained by using our optimized algorithm.

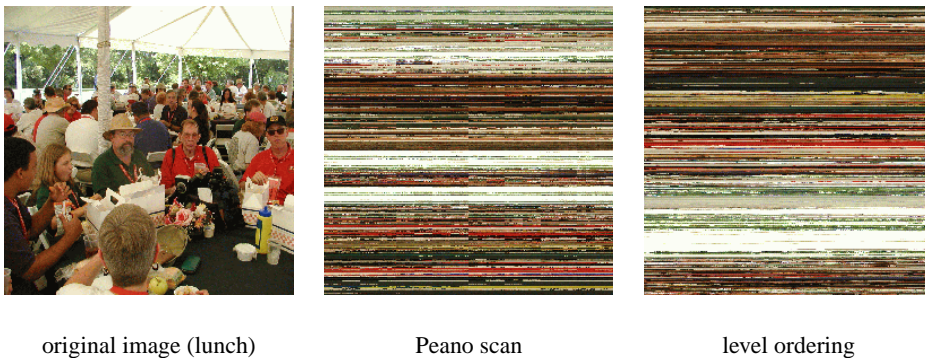


Figure 8: Comparison between three scan techniques. As can be seen, using our algorithm reduces the number of vertical edges. For example, several of the separate white regions in the Peano image are combined in the result generated by our algorithm, and the same holds for the black pixels. This allows the LZ compression algorithm to generate much longer dictionary entries, resulting in an improved compression.

account the mapping size, this 10-20% advantage changed to an increase of 5-10% in the total size. Compressing images obtained from our optimized algorithm resulted in a smaller reduction of 1-10 percentage points. However, for this algorithm the mapping size was much smaller (1/16 of the original mapping size), resulting in a combined size that was up to 9 percentage points smaller when compared to the line scan compression and up to 4 percentage points smaller when compared to the Peano scanned image. Note that the average compression rate for all these images was in the order of 25%. Thus, the improvements achieved by our algorithm increased the compression rate by up to 23%, which is significant, especially in the field of lossless compression.

We have tried to characterize the images for which our algorithm achieves the best improvements. First, the more complex the image the better our algorithm works when compared to other scan methods. Since our algorithm is able to connect remote regions in the image, it can generate long sequences of similar characters, even if such sequences are not present in the original image. In contrast, Peano or line scan methods rely on the local similarity, which is less effective for highly complex images. Second, we have found that, on average, our algorithm achieves better results on larger images. The larger the image, the more likely it is to have similar regions that are not connected. Our algorithm then

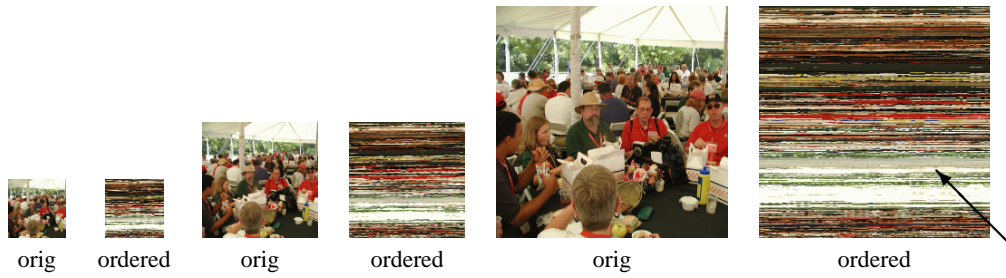


Figure 9: Comparison between the original and reordered images on different levels of the quadtree. Since we hold the upper levels fixed when ordering lower levels, the overall structure of the reordered image does not change much at lower levels. However, local changes can still be made, and they result in a smoother image. For example, the white segment (see arrow) in the rightmost figure is an ordered version of the middle figure.



Figure 10: The rest of the images used for the comparison in Table 7

reorders the image such that these regions are connected, resulting in an improved compression.

8. Conclusions and Future Work

In this paper we have introduced a new algorithm for context-based pixel scan in image space. Our algorithm works on a hierarchical representation of the input image, by optimally reordering levels in the corresponding quadtree. The reordered image is smoother than the input image, resulting in an increased autocorrelation of the pixel sequence. Our algorithm is fast and efficient, working in $O(n)$ time and space, and thus can be applied to arbitrary large images.

When an image is reordered, one must encode the new ordering in order to be able to retrieve the original image. This is the overhead associated with any context-based image ordering. To reduce the size of the encoding, we have presented an optimized version of our algorithm which does not directly encode the ordering of low levels in the tree. Instead, we *learn* this order from the image itself, using upper levels in the tree that are encoded. This allows us, for the first time, to present a context-based ordering algorithm that improves lossless image compression. We have shown that our algorithm improves GIF compression rates by up to 9% when compared to the line scan result and up to 4% compared to the Peano scan result. We have also characterized the images for which our algorithm provides the best increase in compression rates.

There are a number of ways in which this work can be extended. First, we would like to further reduce the compression differences between our original and optimized algorithms, without a large increase in the encoding size. Instead of not encoding entire levels, we can decide whether to encode a specific subtree or not, depending on the improvement we get from this encoding. To make this decision, we need a metric to evaluate the decrease in compression rate with respect to the increase of our goal function (sum of distances). Another direction is to extend our ordering algorithm so that it can be used with other lossless image compression algorithms. While GIF is more suitable for indexed-color images, the lossless jpeg¹² works well on continuous tone images. However, unlike GIF, this algorithm works on the two dimensional context of the image, and thus requires a two-dimensional ordering of the quadtree. Unfortunately, even the restricted version of this problem is NP-hard. Thus, we are interested in finding useful approximation algorithms for this problem.

Acknowledgements

We thank Tommi Jaakkola, David Gifford and Renato Pajarola for useful discussions and for reviewing an early version of this work. Z.B.J. is supported by a Fellowship from the Program in Mathematics and Molecular Biology at the Florida State University, with funding from the Burroughs

Wellcome Fund Interfaces Program. D.C.O was supported in part by the Israel Science Foundation founded by the Israel Academy of Sciences and Humanities, and by the Israeli Ministry of Science, and by a grant from the German Israel Foundation (GIF) and was partially supported by the EU research project 'Multiresolution in Geometric Modelling (MINGLE)' under grant HPRN-CT-1999-00117.

References

1. A. Ansari and A. Fineberg. Image data compression and ordering using Peano Hilbert scan and lot. *IEEE Transactions on Consumer Electronics*, 38:436–445, 1992.
2. Z. Bar-Joseph, E. Demaine, D. Gifford, T. Jaakkola, and N. Srebro. k -ary clustering with optimal leaf ordering for gene expression data. In *2nd Workshop on Algorithms in Bioinformatics (WABI), LNCS 2542*, pages 506–520, 2002.
3. Z. Bar-Joseph, D. Gifford, and T. Jaakkola. Fast optimal leaf ordering for hierarchical clustering. *Bioinformatics*, 17:s22–s29, 2001.
4. R. Dafner, D. Cohen-Or, and Y. Matias. Context based space filling curves. *Computer Graphics Forum*, 19:209–218, 2000.
5. J. S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Computer Graphics*, pages 361–368. ACM SIGGRAPH, 1997.
6. Y. Fisher. *Fractal Image Compression: Theory and Application*. Springer Verlag, 1995.
7. A. Lempel and J. Ziv. Compression of individual sequences via variable rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.
8. A. Lempel and J. Ziv. Compression of two-dimensional data. *IEEE Transactions on Information Theory*, 32:2–8, 1986.
9. N. Memon, D. Neuhoff, and S. Shende. An analysis of scanning techniques for lossless image coding. *IEEE Transactions on Image Processing*, 9:1837–48, 2000.
10. F. Pinciroli. A Peano-Hilbert derived algorithm for compression of angiocardigraphic images. In *18th Annual Conference on Computers in Cardiology*, 1992.
11. Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.
12. M. Weinberger, G. Seroussi, and G. Sapiro. The loci lossless image compression algorithm: Principles and standardization into jpeg-ls. *IEEE Transactions on Image Processing*, 9:1309–24, 2000.