Lossless Compression of Color Map Images by Context Tree Modeling

Alexander Akimov, Alexander Kolesnikov, and Pasi Fränti

***Abstract*—Significant lossless compression results of color map images have been obtained by dividing the color maps into layers and by compressing the binary layers separately using an optimized context tree model that exploits interlayer dependencies. Even though the use of a binary alphabet simplifies the context tree construction and exploits spatial dependencies efficiently, it is expected that an equivalent or better result would be obtained by operating directly on the color image without layer separation. In this paper, we extend the previous context-tree-based method to operate on color values instead of binary layers. We first generate an n-ary context tree by constructing a complete tree up to a predefined depth, and then prune out nodes that do not provide compression improvements. Experiments show that the proposed method outperforms existing methods for a large set of different color map images.**

***Index Terms*—Context tree compression, lossless image coding, map image coding.**

I. Introduction

In this paper, we consider the problem of lossless compression of raster map images. These types of images usually have few colors, a lot of detail, and are large. An example of a map image is shown in Fig. 1. Predictive coding techniques such as JPEG-LS [1], CALIC [2], [3], TMW [4], and FELICS [5] work well on photographic images with smooth changes in color, but are less efficient on map images, due to the map images' sharp change in colors.

The *CompuServe Graphics Interchange Format* (GIF) and *Portable Network Graphic* (PNG) formats are the most commonly used file formats for compressing graphics. GIF uses LZW compression algorithm [6]. PNG uses the *DEFLATE* algorithm [7], which is a combination of the LZ77 dictionary-based compression algorithm [8] and the Huffman coding. Both of these methods can also be used for the compression of map images. These algorithms are looser than newer algorithms, which are based on context modeling.

Typical map images have high spatial resolution for representing fine details, such as text and graphics objects, but do not have as many color tones as photographic images. The *Piecewise-constant* (PWC) algorithm [9] has been developed for the compression of palette images. It uses two-pass object-based modeling. In the first pass, boundaries between constant color pieces are established by the edge model and encoded according to the edge context model, as proposed by Tate [10]. The color of the pieces are determined and coded in the second pass by finding diagonal connectivity and by color guessing. Finally, an arithmetic coder encodes the resulting information. The latest version of PWC, which includes the *skip-innovation* technique and the streaming single-pass variant [9], remains to be one of the best compression algorithms for palette images.

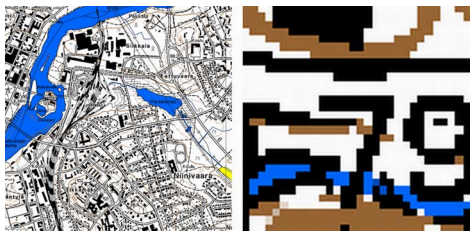

Fig. 1. Example of color map image: full size (left) 1024 × 1024 pixels and (right) 100 × 100 part.

Statistical context modeling that exploits 2-D spatial dependencies has also been applied for lossless palette image compression. The known schemes can be categorized into those that divide the images into binary layers and those that apply context modeling directly to the original colors. The separation of the input image can be done through *color separation* or through *semantic separation* [11], [12]. The binary layers are then compressed by a context-modeling scheme, such as JBIG [13], or by using the *context tree* [14]. The best results for this approach have been achieved by context tree compression with semantic separation [11], [12], but this requires that the encoder has the semantic decomposition available beforehand, which is not generally the case. In terms of color separation, the best results have been achieved by the multilayer context tree (MCT) compression with an optimal order of layers and template pixels [12]. The drawback of this approach is the compression time, which can be quite long due to the time required for the optimal ordering of layers.

A possible alternative to color separation is the separation of the colors into bit planes followed by the separate compression of each bit plane. *Embedded image-domain adaptive compression of simple images* (EIDAC) [15] uses a 3-D context model tailored for the compression of grayscale images. The algorithm divides the image into bit planes and compresses them separately. However, the context pixels are selected not only from the current bit plane, but also from the already processed bit planes.

Manuscript received October 26, 2005 ; revised July 12, 2006. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Bruno Carpentieri.

The authors are with the University of Joensuu, FI-80101 Joensuu, Finland (e-mail: akimov@cs.joensuu.fi; koles@cs.joensuu.fi; franti@cs.joensuu.fi).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIP.2006.887721

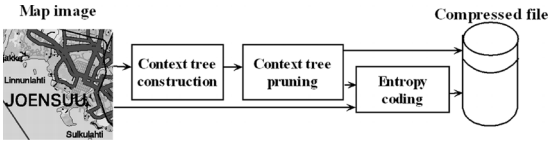

Fig. 2. Overall scheme of the proposed algorithm.

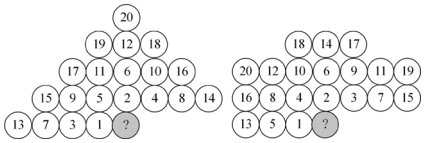

Fig. 3. Default location and order of the neighbor pixels for standard (left) 1-norm and (right) 2-norm templates.

Another approach is to operate directly on the color values. Statistical context-based compression known as the *prediction by partial matching* (PPM) has been applied for the compression of map images [16]. The method is a 2-D version of the original PPM method [17]; it combines a 2-D template with the standard PPM coding. Spatial context modeling is applied to the original colors without any separation into binary layers. The method has been applied both to palette images and street maps [16]. The major problem with PPM-based methods is the *context dilution* problem, which occurs when pixel statistics are distributed over too many contexts, thus degrading the efficiency of the compression.

We propose the *generalized context tree* (GCT) algorithm [18] of n-ary tree with *incomplete structure*. The GCT approach has difficulties in its implementation due to its substantial time and memory requirements, especially for the construction of an optimal incomplete n-ary tree. We propose a fast suboptimal pruning algorithm, which significantly decreases the processing time. The compression consists of two main phases. In the first phase, we construct and prune the context tree. We build up the context tree to a predefined maximum depth and collect the statistics for each node in the tree, and then prune out nodes that do not provide improvement in compression. In the second phase, entropy coding is applied to the image using the optimized context tree. We need to store the context tree into the compressed file, as well. It consists of two parts: the description of the context tree structure and the encoded image. The proposed algorithm is outlined in Fig. 2.

II. Context Tree Modelling

A. Finite Context Modeling

In *context modeling*, the probability of the current pixel U depends on the combination of m already encoded pixels $x^1, \ldots, x^m$. The combination of these pixel values is called *context*. The probabilities of the pixels, generated under the given context, are usually treated as being independent [19]. In 2-D modeling, the context is defined by the set of closest pixels. There are several ways to define the location and the order of the context pixels [19], [20]. Simple examples of a 2-D template are shown in Fig. 3.

The context model is a collection of independent sources of random variables. By the assumption of independence, it is simple to assign probabilities to each new pixel generated in the current context. We denote the frequency of the pixel value k in the context $x^1, \ldots, x^m$ as

$$n_k(x^1, \ldots, x^m) = n(U = k \,|\, x^1, \ldots, x^m). \tag{1}$$

The conditional probability of the pixel value $U = k$, $k \in [1, \ldots, \alpha]$, where α is the number of colors in the image, can then be calculated as

$$p\left(U = k \,|\, x^1, \ldots, x^m\right) = \frac{n_k(x^1, \ldots, x^m)}{\sum_{j=1}^{\alpha} n_j(x^1, \ldots, x^m)}. \tag{2}$$

Using the given statistical model, the entropy coder does the encoding. The adaptive probability estimator of the entropy coder operates according to the formula

$$p\left(U = k \,|\, x^1, \ldots, x^m\right) = \frac{n_k(x^1, \ldots, x^m) + \varepsilon}{\sum_{j=1}^{\alpha} n_j(x^1, \ldots, x^m) + \alpha \cdot \varepsilon}. \tag{3}$$

Here, the parameter ε is used for measuring the uncertainty of the model, and its value depends on the selected modeling scheme [21]. At the beginning of the encoding, we set ε to $1/\alpha$, by analogy with [22].

B. Context Tree Algorithm

Theoretically, a better probability estimation of pixels can be obtained by using a larger context template. However, the number of contexts grows exponentially with the size of the template, and the distribution of the pixel statistics over too many contexts degrades the compression efficiency.

The use of the *context tree algorithm* [14] provides a more efficient approach for context modeling, so that a larger number of neighboring pixels can be taken into account without context dilution. The context tree algorithm is applied for the compression in the same manner as the fixed size context, but with a different context selection. The selection is made by traversing the context tree from the root to a terminal node, each time selecting the branch according to the corresponding pixel value. The terminal node points to the statistical model that is to be used.

Single-pass context tree modeling [14] constructs the context tree adaptively. It makes the selection of the context according to the estimation of its proportion in the reduction of the conditional entropy. If this value outperforms the cost of the node, then it is selected.

The two-pass context tree modeling [14], [20] constructs the tree structure and collects the statistics for each context before coding. The context tree is then pruned in order to minimize the sum of the overall conditional entropy and tree description cost. In this approach, the context selection is done by traversing the context tree until the corresponding symbol points to a nonexisting branch, or until the current node is a leaf.

We use the second approach for constructing the context tree: optimize the context model according to the encoded data and store it to the compressed file. This approach requires a lot of memory and calculation resources during the encoding, but the

decoding is much faster and requires significantly less memory resources, because the tree already exists.

C. Construction of an Initial Context Tree

To construct an initial context tree, it is necessary to process the image data to collect statistics for all potential contexts—that is, for all leaves and internal nodes. Each node stores information of the counts of each color appearing in the particular context. The algorithm of the context tree construction operates as follows.

Step 1) Create a root of the tree.

Step 2) or each pixel U_i, $i \in [1 \dots n]$.

- Traverse the tree along the path defined by the values of the context pixels x_j, $j \in [1 \dots m]$, where the positions of the pixels are defined according to a predefined template.
- If the position of some pixel in the context is outside of the image, then the pixel value is set to zero.
- If some node along the path does not have a consequent branch for the transition to the next context pixel, then the algorithm creates the necessary child node and processes it. Each new node has α counters, which all are initially set to zero.
- In all visited nodes, the algorithm increases the count of the current pixel U_i value by 1.

This completes the construction of the context tree for all possible contexts. The time complexity of the algorithm is $O(m{\cdot}n)$, where m is the maximum depth of the context tree, and n is the number of pixels in the image.

D. Pruning the Context Tree

The initial context tree is pruned by comparing every parent node against its children nodes to find the optimal combination of siblings. We denote the overall tree as T, and the nodes of the tree as w. The number of bits, required to store the node w in the compressed file, is denoted as $c(w)$ and is defined by

$$c(w) = \begin{cases} 1, & \text{if } w \text{ is a leaf} \\ \alpha + 1, & \text{otherwise.} \end{cases} \tag{4}$$

The leaves a significant part of all nodes in the context tree, and (4) reduces the total number of bits required for the context tree description. We denote the set of all terminal nodes of the tree T as $S(T)$. We denote the count of the symbol i as $n_i(s)$, where $s \in S(T)$. The estimated code length generated by a terminal node s is calculated using the following expression [19], [23]:

$$c_T\left(n_1(s), \dots, n_\alpha(s)\right) = -\log_2 \frac{\prod_{i=1}^{\alpha} \prod_{j=0}^{n_i(s)-1} (j+\varepsilon)}{\prod_{j=0}^{n_0(s)+n_1(s)+\cdots+n_\alpha(s)-1} (j+\alpha\cdot\varepsilon)}. \tag{5}$$

This definition corresponds to the result obtained by a single-pass arithmetic coder [21]. We define the cost of the context tree T as

$$L(T) = \sum_{w \in T} c(w) + \sum_{s \in S(T)} c_T\left(n_1(s), n_2(s), \dots, n_\alpha(s)\right). \tag{6}$$

The first term gives the cost of the storage of the tree, and the second term gives the cost of the compression of the image with this tree. The goal of the tree pruning is to modify the structure of the context tree so that the cost function (6) will be minimized. For solving this problem, we use a bottom-up algorithm [21], which is based on the principle that the optimal tree consists of optimal subtrees.

For any node w in the tree T, we denote the vector of counts as $n(w) = (n_1(w), \dots, n_\alpha(w))$, and the child nodes as w_i. We denote the vector describing the structure of the node branches as the *node configuration vector*. This vector $v = (v_1, \dots, v_\alpha)$, $v_i \in \{0, 1\}$, defines which branches will be pruned out in the optimization: If $v_i = 0$, then the i -th branch is pruned.

The maximum number of possible configuration vectors for a node is 2^α. The optimal cost $L_{\text{opt}}(T)$ for any given tree T can be expressed by the recursive (7) and (8)

$$L_{\text{opt}}(T) = \begin{cases} c_T(n(w)) + 1, & \text{if } T \text{ has no subtrees} \\ \min\limits_{v} \{L_v(T, v)\}, & \text{otherwise} \end{cases} \tag{7}$$

$$L_v(T, v) = c_T\left(n(w) - v \circ \left(\sum_i n(w_i)\right)\right) + \sum_i \left(v_i \cdot L_{\text{opt}}\left(T_i\right)\right) + \alpha + 1. \tag{8}$$

Here, $T_i \subset T$ is a subtree of T starting from its child node w_i. The operator "$\circ$" denotes the *Hadamard product* (the element by element product of two vectors/matrices). These formulae require that, for the calculation of the optimal cost of any tree, we first need to calculate the optimal costs of all its subtrees. The calculation of the cost function $L_{\text{opt}}(T)$ and pruning of the context tree T can be described as follows.

Step 1) If T has no subtrees, then return the accumulated code length of its root according to (6).

Step 2) For all subtrees T_i, calculate their optimal costs $L_{\text{opt}}(T_i)$ recursively.

Step 3) According to the found values $L_{\text{opt}}(T_i)$, the vectors of counts $n(t)$ and $n(t_1), \dots, n(t_\alpha)$, find the configuration vector v that minimizes (8).

Step 4) Prune out the subtrees according to the vector v.

Step 5) Return the value $L_v(T, v)$.

The algorithm recursively prunes out all unnecessary branches, and outputs the structure of the optimal context tree. An example of pruning a single node is shown in Fig. 4. The best configuration is chosen between 16 different variants of the pixel distribution between the parent and children. The resulting distribution produces the smallest value of the function (6).

III. Finding the Optimal Configuration Vector

Finding the optimal node configuration vector is the most time-consuming phase in the construction of the α-ary incomplete context tree. In the case of the full context tree, the configuration can be chosen only from two alternatives: either prune all subtrees of the considered node or preserve them all. In the case of incomplete context tree, however, we need to solve a more complicated optimization problem.

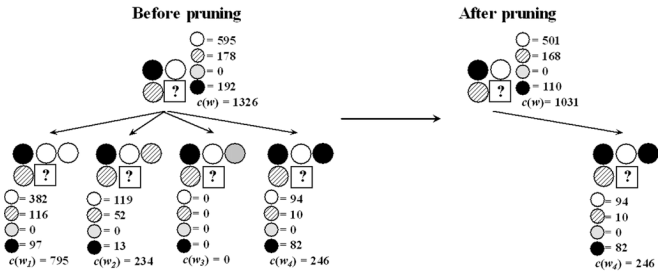

Fig. 4. Example of a single-node pruning: Resulted node configuration is (0,0,0,1).

A. Full Search Approach

We need to process the pruning of each node of the context tree. A straightforward approach is to calculate all possible variants of the subtree configurations and then choose the best one. If the number of nodes in the context tree is N, then the time complexity of the full search is $O(2^{\alpha} \cdot N)$. In practice, the tree pruning requires fewer computations because the number of existing subtrees at each node is usually smaller than α in real map images. Nevertheless, this part is the bottleneck of the algorithm because the pruning can take several hours, even for a small map image.

B. Steepest Descent Approach

One possible way to reduce the time complexity is to compromise the optimality by considering only a small amount of all possible configuration vectors. We apply the well-known *steepest descent* optimization algorithm. According to (7) and (8)m the optimization problem for tree T can be formulated as

$$v_{\min} = \arg \min_{v \in C^{\alpha}} \{L_v(T, v)\}. \tag{9}$$

The candidate solutions $\{v\}$ are considered as the vertices of an α -dimensional hypercube C^{α}.

The proposed optimization algorithm is applied for each node of the context tree. The result of the optimization is the optimal configuration vector and the cost of the node. The algorithm works as follows.

Step 1: Find the starting point of the search.

Step 1.1: Calculate values $L_0 = L_v(T, v = (0, 0, \ldots, 0))$ and $L_1 = L_v(T, v = (1, 1, \ldots, 1))$. Set the start value $L^{\text{start}} = \min\{L_0, L_1\}$.

Step 1.2: If $L^{\text{start}} = L_0$, then the starting point $v^{\text{start}} = (0, 0, \ldots, 0)$, the search direction $\Delta = +1$. Otherwise, the starting point $v^{\text{start}} = (1, 1, \ldots, 1)$ and $\Delta = -1$.

Step 1.3: Set the *left bound* (*LB*) of the search to 1.

Step 2: Process steepest descent optimization for input arguments L^{start}, v^{start} and *LB*.

Step 2.1: If $LB > \alpha$, then return v^{start} and L^{start} as the result of the optimization.

Step 2.2: Generate the set of candidate solutions v^j, $j \in [LB, \ldots, \alpha] : v^j = \{v_1^{\text{start}}, \ldots, v_j^{\text{start}} + \Delta, \ldots, v_{\alpha}^{\text{start}}\}$, $v^j \in C^{\alpha}$. Find value $L^* = \min\{L_v(T, v^j)\}$.

If $L^* \geq L^{\text{start}}$ then return v^{start} and L^{start}.

Step 2.3: Recursively call the optimization Step 2 for each candidate solution v^k, which satisfies: $L_v(T, v^k) - L^* \leq \text{threshold}$, with new input arguments: $L^{\text{start}} = L^*$, $v^{\text{start}} = v^k$ and $LB = k + 1$. Denote the returned results of optimization as v^{*k} and L^{*k} for each v^k.

Step 2.4: Find values $L_{\min} = \min\{L^{*k}\}$ and $v_{\min} = \text{argmin}\{L^{*k}\}$. Return $v_{\min}$ and $L_{\min}$ as the result of the optimization.

Step 3: If $L_{\min} < L^{\text{start}}$, then set v^{start} to $v_{\min}$, L^{start} to $L_{\min}$, *LB* to 1, and process the Step 2 of the algorithm again.

```
[ResultVector, ResultValue] ← OptimalConfiguration(Node)
Begin

   Vector0 ← {0,0,…,0};
   Value0  ← EstimateCodeLength(Node, Vector0);
   Vector1 ← {1,1,…,1};
   Value1  ← EstimateCodeLength(Node, Vector1);

   if Value0 < Value1 then
      StartVector ← Vector0;
      StartValue  ← Value0;
      delta ← +1;
   else
      StartVector ← Vector1;
      StartValue  ← Value1;
      delta ← -1;
   endif

   [ResultVector, ResultValue] ←
      SteepestDescent(Node, StartVector, StartValue, delta, 1);

  if ResultValue < StartValue then
     StartVector ← ResultVector;
     StartValue  ← ResultValue;
     [ResultVector, ResultValue] ←
       SteepestDescent(Node, StartVector, StartValue, delta, 1);
  endif

End.
```

Fig. 5. Pseudocode of the local optimal configuration search.

The pseudocodes of the proposed optimization technique and the steepest descent algorithm are shown in Figs. 5 and 6, correspondingly. In the worst case, the number of calculations in this steepest descendent algorithm for each node is 2^{α}, which is the same as in the full search. However, we can adjust the tradeoff between time and optimality of the algorithm by a suitable selection of the threshold. The threshold value defines how large the set of possible solutions in the steepest descent optimization is. If the threshold value is large, then the set of solutions is large and the result of the optimization is close to the global optimum, but the algorithm works slower. A small threshold value narrows the set of processed solutions, thereby increasing the speed of algorithm and reducing the accuracy of the optimization. We use the threshold value set to 0.01, which was found experimentally.

IV. Hybrid Tree Variant

Free tree coding was introduced in [20]. The locations of the context pixels in the previous context algorithm are defined by a static template, whereas the free tree optimizes the locations to the encoded image. The locations of each context pixel depend on the values of the previous context pixels. Fig. 7 shows an example of the binary free tree with optimized pixels locations. In the example, the coordinates of the second context pixels depend on the value of the pixel with relative coordinates $(-1, 0)$. If the value of the pixel is white, then the next context pixel is located at position $(-1, -1)$. Otherwise, the next context pixel is located at position $(-2, 0)$.

A greedy algorithm for free tree constructing has been described in [19]. The algorithm builds up the free tree level-by-level, proceeding through the entire image during each iteration.

```
[ResultVector, ResultValue] ←
 SteepestDescent(Node, StartVector, StartValue, delta, LeftBound)
Begin
  Min ← StartValue;
  ResultVector ← StartVector;
  ResultValue  ← StartValue;

  for i:= LeftBound to NColors
    LocalVector    ← StartVector;
    LocalValue[i]  ← StartValue;
    if LocalVector[i]+delta≥0 and LocalVector[i] + delta≤1
       LocalVector[i]  ← LocalVector[i] + delta;
       LocalValue[i]← EstimateCodeLength(Node, LocalVector);
    endif
    if LocalValue[i] < Min then
       Min := LocalValue[i];
    endif
  endfor

  if StartValue = Min then
   return;
  endif

  for i:= LeftBound to NColors
   if LocalValue[i] - Min ≤ threshold then
     LocalVector    ← StartVector;
     LocalVector[i] ← LocalVector[i] + delta;
     [TempVector, TempValue] ←
          SteepestDescent(Node, LocalVector[i], Min, delta, i+1);
     if TempValue < ResultValue then
        ResultVector  ← TempVector;
        ResultValue   ← TempValue;
     endif
   endif
  endfor

End.
```

Fig. 6. Pseudocode of the recursive steepest descent algorithm.

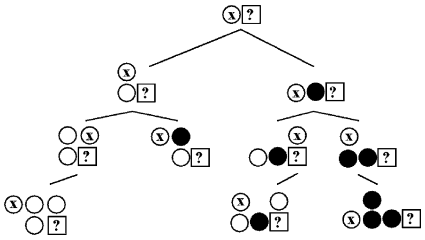

Fig. 7. Illustrative example of the free tree.

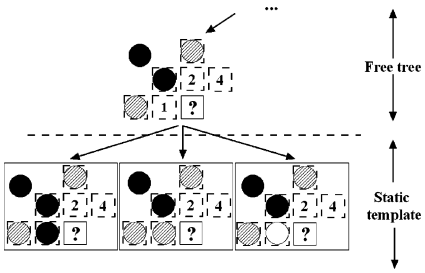

Fig. 8. Example of the hybrid tree: Locations of depths less than or equal to four are defined by the free tree, and locations of bigger depth are defined by unused positions in the static template.

The number of all possible contexts and memory requirements increase exponentially with the depth of the free tree. Therefore, the construction of a free tree that is deep can be problematic.

As an alternative to the free tree, we consider a so-called *hybrid tree,* where the free tree is built up only to a predefined depth f, and the locations of deeper contexts are defined by a fixed template. During encoding we traverse along the context tree and mark all locations that have occurred in a fixed template. For contexts with depths smaller than or equal to f, we choose the locations according to the free tree structure. For contexts with depth greater than f, we choose the first unused location in the fixed predefined template. Fig. 8 shows an example of the hybrid tree.

The hybrid tree coding produces better compression than the fixed one, but the construction of the tree and the procedure for choosing the locations significantly increase the processing time.

Fig. 9. Sample 256 × 256 pixel fragments of the test images.

TABLE I
PROPERTIES OF THE MAP IMAGES FROM DIFFERENT TEST SETS

	Scale	Type of map	No. of Images	Image size	No. of colors
Set #1	1 : 20 000	Topographic	5	5000×5000	6
Set #2	1 : 8 000	Topographic	4	1024×1024	7
Set #3	1 : 100 000	Roads	4	1024×1024	16
Set #4	1 : 800 000	Roads	4	1024×1024	16
Set #5	1 : 250 000	City roads	2	800×800	16
Set #6	1 : 400 000	Roads	5	1250×1250	67

V. EXPERIMENTS AND DISCUSSIONS

The proposed algorithm was tested on six sets of different map images; see Fig. 9 for illustrative examples and Table I for their statistics. The sets from #1 to #4 are from the database of the National Land Survey of Finland [24]. We compare the following compression methods.

- GIF: CompuServe interchange format [6].
- PNG: Portable network graphics format [7], [8].
- MCT: Multilayer binary context tree with optimized order of the layers [12].
- PWC: Piecewise-constant image model [9].
- CT: The n-ary context tree modeling with full tree structure [20], [25].
- GCT: Generalized context tree algorithm with an incomplete n-ary tree structure with fixed template.
- GCT-HT: Generalized context tree algorithm with incomplete n-ary tree structure with hybrid tree.

We used the MQ coder as the entropy coder, which is a modification of the Q coder [26]. All benchmarking was done on a 3-GHz P4 computer, with 1-GB of RAM, under Windows XP.

The MCT algorithm was applied to the binary layers after color separation of the test images. The rest of the algorithms were applied to the original color images. Because of the huge processing time needed for the optimal ordering of the layers, the MCT algorithm was run only with the first five test sets.

TABLE II
COMPRESSION RESULTS (BITS PER PIXEL)

	GIF	PNG	MCT	PWC	CT	GCT	GCT-HT
Set #1	0.557	0.830	0.162	0.236	0.172	0.153	0.152
Set #2	0.670	0.705	0.252	0.268	0.278	0.248	0.223
Set #3	2.125	2.067	1.416	1.436	1.137	1.108	1.087
Set #4	2.121	2.059	1.512	1.343	1.141	1.090	1.048
Set #5	1.866	1.785	1.371	1.221	1.079	1.037	0.992
Average:	1.468	1.489	0.943	0.901	0.761	0.727	0.700
Set # 6	0.986	0.845	N/A	0.355	0.376	0.340	0.319
Average:	1.386	1.382	N/A	0.814	0.697	0.663	0.637

TABLE III
TOTAL PROCESSING TIMES (IN SECONDS) OF THE GCT FOR DIFFERENT STEPS OF THE COMPRESSION AND DECOMPRESSION

	Depth of the context tree	Compression		Decompression
		Tree construction	Entropy encoding	
Set #1	22	893.78	83.66	90.31
Set #2	22	41.55	1.91	2.13
Set #3	6	31.22	1.47	2.34
Set #4	6	42.91	1.48	1.86
Set #5	6	13.22	0.41	0.98
Set #6	8	99.42	2.91	3.86

We also implemented the two-pass version of CT algorithm [14], [20] with backward pruning [20]. It utilizes variable depth context modeling with a full n-ary context tree.

Compression results are summarized in Table II. The proposed algorithm outperformed all comparative methods in terms of compression performance. The GCT algorithm worked better with all test sets because it utilized the color dependencies better. The proposed algorithm gave at least a 6% lower bit rate, on average, than the comparative methods. Between the two variants, the hybrid tree approach (GCT-HT) was slightly better than the GCT using a static template.

Tables III and IV report the processing times of the GCT and GCT-HT algorithms. In the compression stage, most time was spent on context tree construction and pruning. The experiments show that the algorithm is asymmetric in execution time: the decompression stage takes much less time than the compression stage. It can be observed that the GCT method is suitable for online processing of images of reasonably small size.

Table V shows the performance of the steepest descent approach in comparison with the full search for a single map from test set #5. The results indicate that the steepest descending approach provides results almost as good the full search but is significantly faster. It is, therefore, more applicable to the online processing of images.

Fig. 10 shows the dependency of the compression efficiency and the image size. For this experiment, we took the images from test set #5 and divided them into fragments of dimensions 100×100, 200×200, and 400×400 pixels. The resulting bit rates were calculated as the average of all compressed files. The

TABLE IV
TOTAL PROCESSING TIMES (IN SECONDS) FOR DIFFERENT PHASES OF COMPRESSION AND DECOMPRESSION OF THE GCT-HT ALGORITHM

	Depth of the free/hybrid tree	Compression		Decompression
		Tree construction	Entropy encoding	
Set #1	8/22	7427.18	1070.11	1095.91
Set #2	8/22	272.14	4.52	5.97
Set #3	6/6	427.14	2.75	6.55
Set #4	6/6	596.31	2.77	6.83
Set #5	6/6	181.79	1.16	2.48
Set #6	8/8	1410.70	5.70	15.45

TABLE V
COMPRESSION TIMES (IN SECONDS) OF THE GCT FOR FULL SEARCH AND STEEPEST DESCENT APPROACHES AS A FUNCTION OF THE MAXIMUM DEPTH

	Full search			Steepest descent		
Depth	4	6	8	4	6	8
Time (s)	1680.13	2113.32	2561.90	4.53	7.55	17.25
Bit rate	1.537	1.519	1.518	1.540	1.524	1.524

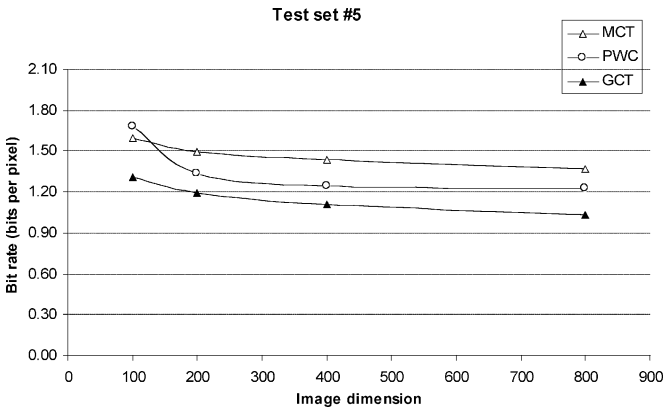

Fig. 10. Dependency of the bit rate on the image size.

experiments showed that the bit rate of the GCT algorithm remains rather stable when operating with images of small size.

Fig. 11 illustrates the dependency of the GCT compression efficiency on the number of colors. The tests were processed on test set #6, where the number of colors was decreased by color quantization from 67 down to 32, 16, 10, 6, and 2.

The proposed algorithm can be used mainly for the compression of palette and halftone images in general, but there are some problems, which can decrease its efficiency in the case of photographic images. The necessity of storing the context tree in the compressed file can decrease the compression performance if the number of colors is increased significantly. The storage demands are about α bits per each node and the space requirement increases exponentially with tree depth. The algorithm is, therefore, not expected to work efficiently for images with a large color palette (more than 128 colors), or for small images (with the size less than 100×100 pixels).

In the case of larger images, the processing time of the algorithm can still be a bottleneck in real time applications. Most of the time, the compression is taken by the construction and pruning of the context tree, and the time increases as the number

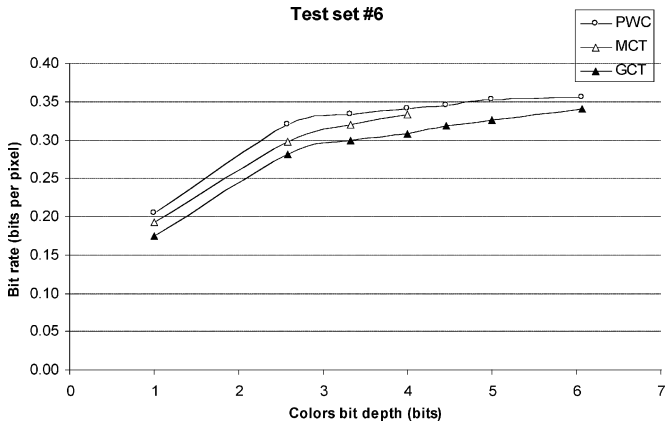

Fig. 11. Dependency of the bit rate on the image color depth.

of colors and maximum depth of the tree increases. The time could be reduced further by applying fast calculation of the estimated code length, in the same manner as proposed in [19].

VI. Conclusion

In this paper, we propose an n-ary context tree model with incomplete tree structure for the lossless compression of color map images. A fast heuristic pruning algorithm was also introduced to decrease the time required in the optimization of the tree structure.

The proposed n-ary incomplete context-tree-based algorithm outperforms the competitive algorithms (MCT, PWC) by 20%, and by 6% in the case of full context tree (CT) algorithm.

The compression method was successfully applied to raster map images up to 67 colors. If the overwhelming memory consumption can be solved in the case of images with a larger number of colors, then it is expected that the method could also be applicable to photographic images. This is a point for further study.

References

[1] M. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS," *IEEE Trans. Image Process.*, vol. 9, no. 8, pp. 1309–1324, Aug. 2000.

[2] X. Wu, "An algorithmic study on lossless image compression," in *Proc. IEEE Data Compression Conf.*, Apr. 1996, pp. 150–159.

[3] N. Memon and A. Venkateswaran, "On ordering color maps for lossless predictive coding," *IEEE Trans. Image Process.*, vol. 5, no. 11, pp. 1522–1527, Nov. 1996.

[4] B. Meyer and P. Tischer, "TMW—A new method for lossless image compression," presented at the Int. Picture Coding Symp., Sep. 1997.

[5] P. Howard and J. Vitter, "Analysis of arithmetic coding for data compression," in *Proc. IEEE Data Compression Conf.*, Apr. 1991, pp. 3–12.

[6] T. Welch, "A technique for high-performance data compression," *Comput. Mag.*, vol. 17, no. 6, pp. 8–19, Jun. 1984.

[7] P. Deutsch, DEFLATE Compressed Data Format Specification, rfc1951, 1996 [Online]. Available: http://www.cis.ohio-state.edu/htbin/rfc/rfc1951.html

[8] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[9] P. Ausbeck, "The piecewise-constant image model," *Proc. IEEE*, vol. 88, no. 11, pp. 1779–1789, Nov. 2000.

[10] S. Tate, "Lossless Compression of Region Edge Maps," Tech. Rep. CS-1992-09, Dept. Computs. Sci., Duke Univ., Durham, NC.

[11] S. Forchhammer and O. Jensen, "Content layer progressive coding of digital maps," *IEEE Trans. Image Process.*, vol. 11, no. 12, pp. 1349–1356, Dec. 2002.

[12] P. Kopylov and P. Fränti, "Compression of map images by multilayer context tree modeling," *IEEE Trans. Image Process.*, vol. 14, no. 1, pp. 1–11, Jan. 2005.

[13] *JBIG, Progressive Bi-Level Image Compression*, ISO/IEC International Standard 11544, 1993.

[14] J. Rissanen, "A universal data compression system," *IEEE Trans. Inf. Theory*, vol. 29, no. 5, pp. 656–664, Sep. 1983.

[15] Y. Yoo, Y. Kwon, and A. Ortega, "Embedded image-domain adaptive compression of simple images," in *Proc. 32nd Asilomar Conf. Signals, Systems, Computers*, Nov. 1998, vol. 2, pp. 1256–1260.

[16] S. Forchhammer and J. Salinas, "Progressive coding of palette images and digital maps," in *Proc. IEEE Data Compression Conf.*, Apr. 2002, pp. 362–371.

[17] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. 32, no. 4, pp. 396–402, Apr. 1984.

[18] A. Martin, G. Seroussi, and M. Weinberger, "Linear time universal coding and time reversal of tree sources via FSM closure," *IEEE Trans. Inf. Theory*, vol. 50, no. 7, pp. 1442–1468, Jul. 2004.

[19] B. Martins and S. Forchhammer, "Tree coding of bi-level images," *IEEE Trans. Image Process.*, vol. 7, no. 4, pp. 517–528, Apr. 1998.

[20] R. Nohre, "Topics in Descriptive Complexity," Ph.D. dissertation, Univ. Linköping, Linköping, Sweden, 1994.

[21] P. Howard and J. Vitter, "Fast and efficient lossless image compression," in *Proc. IEEE Data Compression Conf.*, Apr. 1993, pp. 351–360.

[22] G. Martin, "An algorithm for removing redundancy from a digitized message," presented at the Video and Data Recording Conf., Jul. 1979.

[23] M. Weinberger and J. Rissanen, "A universal finite memory source," *IEEE Trans. Inf. Theory*, vol. 41, no. 5, pp. 643–652, May 1995.

[24] National Land Survey of Finland. Helsinki, Finland [Online]. Available: http://www.nls.fi/index_e.html

[25] M. Weinberger, J. Rissanen, and R. Arps, "Application of universal context modeling to lossless compression of gray-scale images," *IEEE Trans. Image Process.*, vol. 5, no. 4, pp. 575–586, Apr. 1996.

[26] J. Mitchell and W. Pennebaker, "Software implementations of the Q-coder," *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 753–774, Nov. 1988.

Alexander Akimov received the M.Sc. degree in apllied mathematics in 2000 from Saint Petersburg State University, Saint Petersburg, Russia, and the M.Sc degree in computer science in 2001 from the University of Joensuu, Joensuu, Finland, where he is currently pursuing the Ph.D.degree in Department of Computer Science.

His main research areas are the compression of raster and vector map images.

Alexander Kolesnikov received the M.Sc. degree in physics in 1976 from the Novosibirsk State University, U.S.S.R., and the Ph.D. degree in computer science in 2003 from the University of Joensuu, Joensuu, Finland.

From 1976 to 2003, he was a Senior Research Fellow with the Institute of Automation and Electrometry, Russian Academy of Sciences, Novosibirsk, Russia. In 2003, he joined the Department of Computer Science, University of Joensuu. His main research areas are in signal and image processing, vector map processing, and compression.

Pasi Fränti received his M.Sc. and Ph.D. degrees in computer science in 1991 and 1994, respectively, from the University of Turku, Finland.

From 1996 to 1999, he was a Postdoctoral Researcher with the Academy of Finland. Since 2000, he has been a Professor at the University of Joensuu, Joensuu, Finland. His primary research interests are in image compression, clusterization, and speech technology.