Lossless compression of color map images by context tree modeling

Alexander Akimov, Alexander Kolesnikov and Pasi Fränti

***Abstract*— The best lossless compression results of color map images have been obtained by dividing the color maps into layers, and by compressing the binary layers separately by using an optimized context tree model that exploits inter-layer dependencies. Even though the use of binary alphabet simplifies the context tree construction and exploits spatial dependencies efficiently, it is expected that 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 the 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 improvement in compression. Experiments show that the proposed method outperforms existing methods for a large set of different color map images.**

***Index Terms*—Map image coding, context tree compression, lossless image coding.**

I. Introduction

We consider the problem of lossless compression of raster map images. This class of images is characterized by a small number of colors, a lot of structured details, and a large size. An example of a map image is shown in Figure 1. Predictive coding techniques such as JPEG-LS [4], CALIC [5] [6], TMW [24] and FELICS [25] work well on photographic images with smooth changes of colors but are less efficient on map images due to the sharp change of colors.

CompuServe Graphics Interchange Format (GIF) and *Portable Network Graphic* (PNG) formats are the most commonly used file formats for compressing graphics. The first one uses LZW compression algorithm [1]. The second one uses the *DEFLATE* algorithm [2], which is a combination of LZ77 dictionary based compression algorithm [3] and Huffman coding. Both of these methods can also be used for the compression of map images. These algorithms are the oldest ones and loose to newer algorithms, based on context based modeling.

As typical map images have a high spatial resolution for representing fine details such as text and graphics objects but not so much color tones as photographic images. *Piecewise-constant* (PWC) algorithm [14] have been developed for compression. of palette images. It uses a two-pass object-based modeling. In the first pass, the boundaries between constant

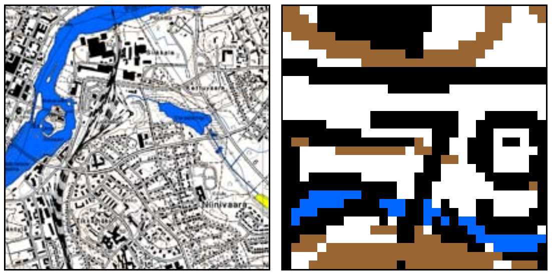
Fig. 1. An example of color map image: full size 1024×1024 pixels (left), and 100×100 part (right).

color pieces are established by the edge model and encoded according to the edge context model, proposed by Tate [24]. The color of the pieces are determined and coded in the second pass by finding diagonal connectivity and color guessing. Finally, an arithmetic coder encodes the resulting information. The latest version of PWC, which includes the *skip-innovation* technique and streaming single-pass variant [14], still remains as one of the best compression algorithm for palette images.

Statistical context modeling that exploits 2-D spatial dependencies is applied for the lossless palette image compression. The known schemes can be categorized to those that divide the images into binary layers, and to those that apply context modeling directly to the original colors. The separation of the input image can be done by *color separation*, or by *semantic separation* [9, 10]. The binary layers are then compressed by a context modeling scheme such as JBIG [7], or by using *context tree* [11]. The best results for this approach have been achieved for context tree compression with semantic separation [9, 10], but this requires that the encoder have the semantic decomposition available beforehand, which is not the case in general. In the case of color separation, best results have been achieved by multilayer context tree (MCT) compression with optimal order of layers and template pixels [10]. The drawback of this approach is the time taken by the compression, which can be quite huge due to the time required by the optimal ordering of layers.

A possible alternative to the color separation is a separation of the colors into bit-planes following by the compression of them separately. *Embedded image-domain adaptive compression of simple images* (EIDAC) [12] uses three dimensional context model tailored for the compression of grayscale images. The algorithm divides the image into bit planes and compresses them separately but the context pixels are selected not only from the current bit plane but also from

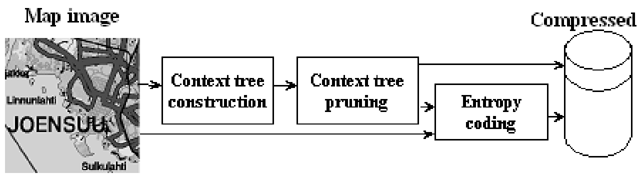

Fig. 2. Overall scheme of the proposed algorithm.

the already processed bit planes.

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 [13]. The method is a 2-D version of the original PPM method by combining a 2-D template with the standard PPM coding. The neighboring context modeling is applied for the original colors without separation into binary layers. The method has been applied both to palette images and street maps [13]. The major problem of the PPM-based methods is the *context dilution* problem, when the pixel statistics are distributed over too many contexts, thus affecting the efficiency of the compression.

We propose a *generalized context tree* (GCT) algorithm with *n*-ary tree with *incomplete structure*. This approach implies difficulties in the implementation due to its great time and memory requirements. Especially the construction of an optimal incomplete *n*-ary tree is problematic. We, therefore, propose a fast sub-optimal heuristic 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, which finally consists of two parts: the description of the context tree structure and the encoded image. The proposed compression algorithm is outlined in Figure 2.

II. CONTEXT TREE MODELLING

A. Finite context modeling

In *context modeling*, the probability of the current pixel U is estimated conditioned on the combination of its m previously encoded pixels $x^1,\dots,x^m$. 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 [17]. In 2-D modeling the context is defined by a set of closest pixels. There are several ways how to define the location and the order of the context pixels [17, 21]. Simple examples of 2-D template are shown in Figure 3.

Thus, 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 at the current context. We denote the frequency of the pixel value k in the context $x^1,\dots,x^m$ as:

$$n_k(x^1,\dots,x^m) = n(U = k \mid x^1,\dots,x^m) \tag{1}$$

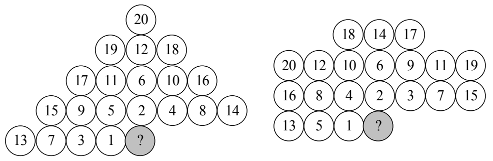

Fig. 3. Default location and order of the neighbor pixels for standard 1-norm (left) and 2-norm (right) templates.

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

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

We assume that an entropy coder makes the encoding of the given statistical model. Adaptive probability estimator of the entropy coder operates by the following formula (3).

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

Here the parameter ε is used for measuring uncertainty of the model, and its value depends on the selected modeling scheme [19]. At the beginning of encoding we set ε to $1/\alpha$, by analogy with [20].

B. Context tree algorithm

Theoretically, 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 affects the compression efficiency.

The use of *context tree* [11] provides a more efficient approach for the context modeling, so that a larger number of neighboring pixels can be taken into account without the context dilution. Context tree is applied for the compression in the same manner as the fixed size context; only the context selection is different. The context selection is made by traversing the context tree from the root to a terminal node, and at each time selecting the branch according to the corresponding previous pixel value. The terminal node points to the statistical model that is to be used.

The single pass context tree modeling [11] makes the selection of the context according to the estimation of its share to the reducing of the conditional entropy. If this value outperforms the cost of the node then it is selected.

The dual pass context tree modeling [11, 21] construct the tree structure and collect the statistics for each context before the entropy encoding. The context tree is 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 non-existing branch, or the current node is a leaf.

We use the second approach for constructing the context tree: optimize the context model according to 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 as the tree already exists.

The tree construction consists of two main phases: initialization of the context tree, and pruning of the constructed tree. These phases will be described below.

C. Construction of an initial context tree

To construct an initial context tree for the input image, we need to process through the image data to collect statistics for all potential contexts: leaves and internal nodes. Each node stores information of the counts of each color appearing in particular context. The algorithm of the context tree construction by processing every pixel in the image as follows:

Step 1: Create a root of the tree.

Step 2: For each pixel U_i, $i\in[1..n]$

- Traverse the tree along the path defined by the values of the context pixel x_j, $j\in[1..m]$, where the positions of the pixels are defined according to the predefined template.
- If the positions of some pixels in the context are outside of the image, then set these pixel values to zero.
- If some node along the path does not have a consequent branch for transition to the next context pixel, then create the necessary child node and process it. Each new node has α counters, which are initially set to zero.
- In all visited nodes, increase 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 for finding the optimal combination of siblings. We denote the overall tree by T, and the nodes of the tree by w. We estimate the number of bits required to store the node w in the compressed file by $c(w)$:

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

The leaves constitute 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 will denote the set of all terminal nodes of the tree T as $S(T)$. We denote the count of the symbol i as $n_i(s)$, $s\in S(T)$. The estimated code length generated by a terminal node s is calculated using the following expression [15, 17]:

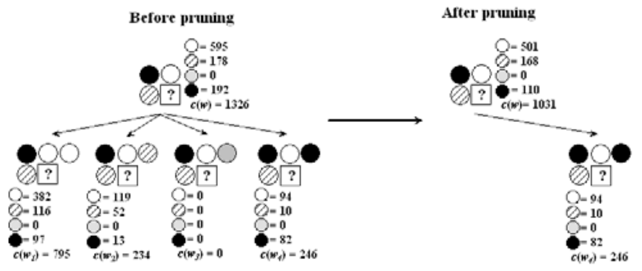

Fig. 4. Example of a single node pruning: resulted node configuration is (0,0,0,1).

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

This definition corresponds to the result obtained by a one-pass arithmetic coder [19]. We define the cost of the context tree T as:

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

The first term gives the cost of the storage of the tree, and the second term the cost of 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 used 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 $n(w)=(n_1(w),\ldots,n_\alpha(w))$ and the child nodes as w_i. We denote the vector describing the structure of node branches as the *node configuration vector*. This vector $v=(v_1,\ldots,v_\alpha)$, $v_i\in\{0,1\}$, defines which branches will be pruned out after the optimization: if $v_i=0$, then the i-th branch is pruned.

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

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

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

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 calculation of the optimal cost of any tree we need firstly to calculate optimal costs of all its subtrees. The calculation of the cost function $L_{opt}(T)$ 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 $L_{opt}(T_i)$ recursively.

```
[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.

```
[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.

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

Step 4: Prune out the subtrees according the found 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 Figure 4. The best configuration was chosen from 16 different variants and resulting distribution of the statistics between parent and children produces 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 from two alternatives only: either prune all subtrees of the considered node, or preserve them all. In the case of incomplete context tree, however, we need to solve more complicated optimization problem.

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 subtrees 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 less 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 small map image.

B. Steepest descent approach

One possible way to reduce the time complexity of the context tree construction is to compromise the optimality by considering only a small part of all possible configuration vectors. We apply the well known *steepest descent* optimization algorithm. According to (7) and (8) 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 α-dimensional hypercube C^α.

The proposed optimization algorithm is called 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 $L_0 = L_v(T, v = (0,0,..,0))$ and $L_1 = L_v(T, v = (1,1,..,1))$. Set the start value $L^{start} = \min\{L_0, L_1\}$.

Step 1.2: If $L^{start} = L_0$, then the starting point $v^{start} = (0,0,\ldots,0)$, the search direction $\Delta = +1$. Otherwise the starting point $v^{start} = (1,1,\ldots,1)$ 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, $j \in [LB,\ldots,\alpha]$: $v^j = \{v_1^{start},\ldots,v_j^{start}+\Delta,\ldots,v_\alpha^{start}\}$, $v^j \in C^\alpha$. Find value $L^* = \min\{L_v(T,v^j)\}$. If $L^* \geq L^{start}$ then return v^{start} and L^{start}.

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

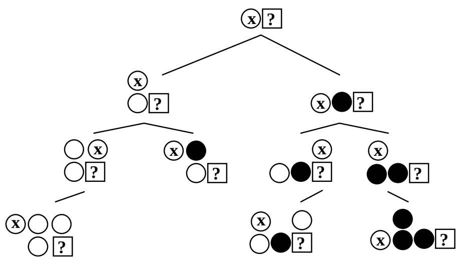

Fig. 7. Illustrative example of the free tree.

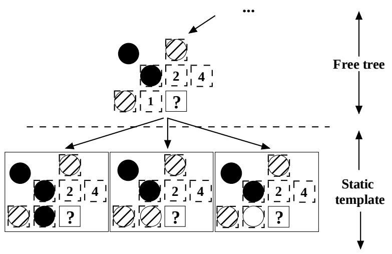

Fig. 8. Example of the hybrid tree: locations of depths less or equal four are defined by free tree and locations of bigger depth are defined by unused positions in static template.

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

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

The pseudocodes of the proposed optimization technique and the steepest descent algorithm are shown in Figures 5 and 6, respectively. 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 is the set of possible solutions in the steepest descent optimization. If the threshold value is large, then the set of solutions is wide and the result of the optimization is close to the global optimum. The same time the algorithm works slower. A small threshold value narrows the set of processed solutions, therefore increases the speed of algorithm and reduces the accuracy of the optimization. We use the threshold value set to 0.01, which was found experimentally.

IV. Hybrid Tree Variant

The *free tree* coding was introduced in [21]. While the locations of context pixels in context algorithm are defined by a static template, the free tree optimizes the locations to the encoded image. The locations of each context pixel depends on the values of previous context pixels. Figure 7 shows an example of the binary free tree with optimized pixels locations. Here, for example, the coordinates of the second context pixels depend on the value of the pixel with relative coordinates (-1, 0). If the pixel's value is white then the next context pixel is located at position (-1,-1). Otherwise, if the pixel is black then the next context pixel is located at position (-2, 0). A greedy algorithm for the free tree constructing has been described in [17]. The algorithm builds up the free tree level by level, processing through the entire image during each iteration. The number of all possible contexts and memory requirements grows up exponentially with increasing of free tree depth. Due this the construction of a deep free tree can be problematic.

The alternative to free tree is a so-called *hybrid tree,* when the free tree is build up only to some 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 occurred locations in a fixed template as used ones. For contexts with depth smaller or equal to *f* we choose locations according the free tree structure. For contexts with depth greater than *f* we choose those unused location in the fixed template, which is the closest to encoded pixel. Figure 8 shows an example of the hybrid tree. The hybrid tree coding produces better compression than the fixed one, but the free tree construction and the procedure of choosing the closest locations significantly increases the processing time.

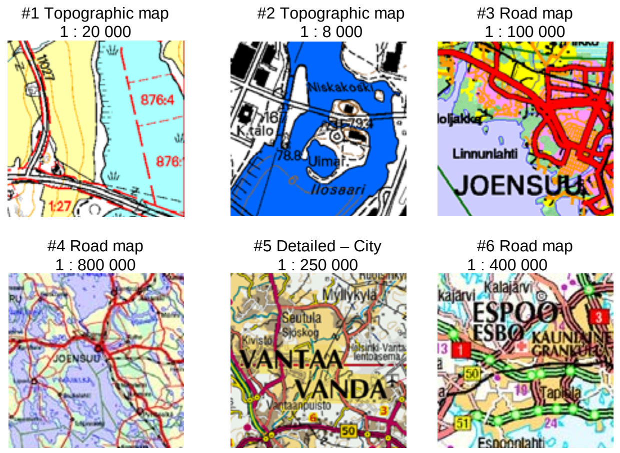

Fig. 9. Sample 256×256 pixel fragments of the test images.

V. Experiments and Discussions

The proposed algorithm was tested on the six sets of different map images, see Figure 9 for illustrative examples, and Table 1 for their statistics. The sets from #1 to #4 are from the database of National Land Survey of Finland [22]. We compare the following compression methods:

- GIF : CompuServe interchange format [1],
- PNG: portable network graphics format [2, 3],
- MCT: multilayer binary context tree with optimized ordering of the layers [10],
- PWC: piecewise-constant image model [14],
- CT: the *n*-ary context tree modeling with full tree structure [16, 21],
- 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.

The compression results are summarized in Table 2. The MCT algorithm is applied for binary layers with color separation of the encoded images. The rest of the algorithms

TABLE I
PROPERTIES OF THE MAP IMAGES FROM DIFFERENT TEST SETS

	Scale	Type of map	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

TABLE 2
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

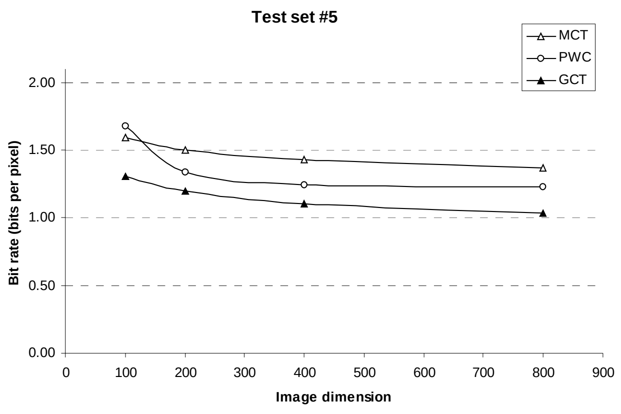

Fig. 10. Dependency of the bit rate on the image size.

are applied to the original color images. The MCT algorithm was run only with the first 5 test sets due to its huge processing time needed for the optimal ordering of the layers.

We implemented the dual pass version of CT algorithm [11, 21] with backward pruning [21]. It utilizes the variable depth context modeling with full *n*-ary context tree.

The proposed algorithm outperforms all comparative methods in terms of compression performance. The GCT algorithm works better with all test sets because it utilizes better the color dependencies. The proposed algorithm gives at least 6% lower bit rate, on average, in comparison to the comparative methods. Among the two variants, the hybrid tree approach (GCT-HT) is slightly better than the GCT using static template.

Tables 3 and 4 report the processing times of the GCT and GCT-HT algorithms. In the compression stage, most time is spent for the context tree construction and pruning. The experiments show that the algorithm is asymmetric in the execution time: decompression takes much less time than the compression stage. It can be observed that the GCT method is

TABLE 3
TOTAL PROCESSING TIMES (SEC) 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

TABLE 4
TOTAL PROCESSING TIMES (SEC) 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 5
COMPRESSION TIMES (SEC) 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

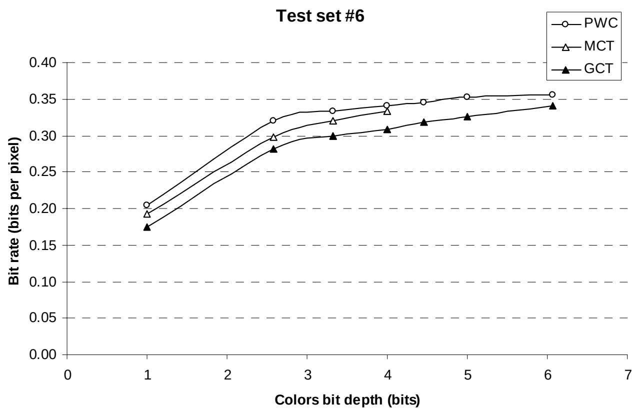

Fig. 11. Dependency of the bit rate on the image color depth.

suitable for on-line processing for images of reasonably small size.

The GCT-HT algorithm builds up the context tree according to the free tree approach until the predefined depth. The static template defines the pixel locations for bigger depths. The hybrid tree approach gives a small improvement in compression performance but makes the compression much slower.

Table 5 shows the performance of the steepest descent approach in comparison with the full search for a single map from the test set #5. We used MQ coder as the entropy coder, which is a modification of Q coder [26]. The results indicate that the steepest descending approach provides almost as good results as the full search but significantly faster. It is therefore better applicable for on-line processing for images of this size. All benchmarking were done on a 3 GHz P4 computer with 1 GB of RAM under Windows XP.

Figure 10 shows the dependency of the compression efficiency and the image size. For this experiment, we took the images from the test set #5, and divided them into fragments of dimensions 100×100, 200×200 and 400×400 pixels. The resulting bit rate was calculated as the average of all compressed files. The experiments show that the bit rate of the GCT algorithm remains rather stable when operating for images of smaller size.

Figure 11 illustrates the dependency of the GCT compression efficiency for different number of colors. The tests were processed on the 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 grows up exponentially with the increase of the tree depth. The algorithm is therefore not expected to work efficiently for images with a large color palette (more than 128 colors), or for really 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 time in the compression is taken by the construction and pruning of the context tree, and the time grows up with the increasing the number of colors and maximum depth of the tree. The time can be reduced further by applying fast calculation of the estimated code length, in the same manner as was proposed in [17].

VI. Conclusions

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) on 20%, and on 6% in comparing with the implemented algorithm, based on full context tree approach (CT).

The method was successfully applied to 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 of further studies.

References

[1] T. Welch, A Technique for high-performance data compression, *Computer Magazine,* vol 17, no. 6, pp. 8-19, June 1984.

[2] P. Deutsch, DEFLATE compressed data format specification, rfc1951, http://www.cis.ohio-state.edu/htbin/rfc/rfc1951.html, May 1996.

[3] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, vol. 23(6), pp. 337-343, May 1977.

[4] M. Weinberger, G. Seroussi, G. Sapiro, The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS, *IEEE Transactions on Image Processing*, vol. 9(8), pp. 1309-1324, August 2000.

[5] X. Wu, An algorithmic study on lossless image compression, *IEEE Proceedings of Data Compression Conference*, pp. 150-159, April 1996.

[6] N. Memon, A. Venkateswaran, On ordering color maps for lossless predictive coding, *IEEE Transactions on Image Processing*, vol. 5(11), pp. 1522-1527, November 1996.

[7] JBIG, Progressive bi-level image compression, *ISO/IEC International Standard* 11544, 1993

[8] J. Cleary, I. Witten, Data compression using adaptive coding and partial string matching, *IEEE Transactions on Communications*, vol. 32(4), pp. 396-402, April 1984.

[9] S. Forchhammer, O. Jensen, Content layer progressive coding of digital maps, *IEEE Transactions on Image Processing*, vol. 11(12), pp. 1349-1356, December 2002.

[10] P. Kopylov and P. Fränti, Compression of map images by multilayer context tree modeling, *IEEE Transactions on Image Processing*, vol. 11(1), pp. 1-11, January 2005.

[11] J. Rissanen, A universal data compression system, *IEEE Transactions on Information Theory*, vol. 29(5), pp. 656-664, September 1983.

[12] Y. Yoo, Y. Kwon, A. Ortega, Embedded image-domain adaptive compression of simple images, *Conference Record of the Thirty-Second Asilomar Conference on Signals, Systems & Computers*, vol. 2, pp. 1256-1260, November 1998.

[13] S. Forchhammer, J. Salinas, Progressive coding of palette images and digital maps, *IEEE Proceedings of Data Compression Conference*, pp. 362-371, April 2002.

[14] P. Ausbeck, The piecewise-constant image model, *Proceedings of the IEEE*, vol. 88 (11), pp. 1779-1789, November 2000.

[15] M. Weinberger, J. Rissanen, A universal finite memory source, *IEEE Transactions on Information Theory,* vol. 41(3), pp. 643-652, May 1995.

[16] M. Weinberger, J. Rissanen, R. Arps, Application of universal context modeling to lossless compression of gray-scale images, *IEEE Transactions on Image Processing*, vol. 5(4), pp. 575-586, April 1996.

[17] B. Martins, S. Forchhammer, Tree coding of bi-level images, *IEEE Transactions on Image Processing,* vol. 7(4), pp. 517-528, April 1998.

[18] P. Kopylov, P. Fränti, Context tree compression of multi-component map images, *IEEE Proceedings of Data Compression Conference*, pp. 212-221, April 2002.

[19] P. Howard, J. Vitter, Analysis of arithmetic coding for data compression, *IEEE Proceedings of Data Compression Conference*, pp. 3-12, April 1991.

[20] G. Martin, An algorithm for removing redundancy from a digitized message, Presented at: *Video and Data Recording Conference*, July 1979.

[21] R. Nohre, *Topics in descriptive complexity*, PhD Thesis, University of Linköping, Sweden, 1994.

[22] National Land Survey of Finland, Opastinsilta 12 C, P.O.Box 84, 00521 Helsinki, Finland. (http://www.nls.fi/index_e.html)

[23] S. Tate, Lossless compression of region edge maps, Duke University Computer Science Technical Report CS-1992-09.

[24] B. Meyer, P. Tischer, TMW—a new method for lossless image compression, *Proceedings of International Picture Coding Symposium*, September 1997.

[25] P. Howard, J. Vitter, Fast and efficient lossless image compression, *IEEE Proceedings of Data Compression Conference*, pp. 351-360, April 1993.

[26] J. Mitchell, W. Pennebaker: Software Implementations of the Q-Coder. *IBM Journal of Research and Development*, vol. 32(6), pp. 753-774, November 1988.