UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

Compression of map images by multi-level context tree modeling

Pavel Kopylov and Pasi Fränti

Report A-2003-3

ACM I.4.2 E.4 G.2.2
ISSN 0789-7316
ISBN 952-458-327-5

COMPRESSION OF MAP IMAGES BY MULTI-LEVEL CONTEXT TREE MODELING

Pavel Kopylov and Pasi Fränti

Department of Computer Science, University of Joensuu
Box 111, FIN-80101 Joensuu, Finland.

Email: franti@cs.joensuu.fi, justas@cs.joensuu.fi

Abstract: We propose a method for compressing color map images by context tree modeling and arithmetic coding. We consider multi-component map images with semantic layer separation, and images that are divided into binary layers by color separation. The key issue in the compression method is the utilization of inter-layer correlations, and to solve the optimal ordering of the layers. The inter-layer dependencies are acquired by optimizing the context tree for every pair of image layers. The resulting cost matrix of the inter-layer dependencies is considered as a directed spanning tree problem, and solved by algorithm based on the Edmond's algorithm for optimum branching, and by the optimal selection and removal of the background color. The proposed method gives 50:1 compression on a set of map images, which is about 50% better than JBIG, and 16% better than the best comparative method.

Keywords: Image compression, lossless, context modeling, map images, layer ordering, spanning tree.

Statistics: 25 pages, 21 figures, 7 tables, 8000 words, 42000 characters.

1. Introduction

Digital maps are usually stored as *vector graphics* in a database for retrieving the data using spatial location as the search key. The visual outlook of maps representing the same region varies depending on the type of the map (*topographic* or *road map*), and on the desired scale (*local* or *regional map*). Vector representation is convenient for zooming as the maps can be displayed in any resolution defined by the user. The maps can be converted to *raster images* for data transmission, distribution via internet, or simply because of incompatibility of the vector representations of different systems. A compressed raster image format provides a reasonable solution in the form of compact storage size and compatible format.

Typical map images have high spatial resolution for representing fine details such as text and graphics objects but not so much color tones as photographic images. Thus, suitable compression method could be found among the lossless *palette image* compression methods such as GIF and PNG. The CompuServe Graphics Interchange Format (GIF) is based on LZW dictionary compressor [1]. Portable Network Graphics (PNG) provides a patent-free replacement for GIF. It encodes the image using the *deflate* [2] algorithm, which is a combination of LZ77 *dictionary compression* [3] and *Huffman coding*.

On the other hand, lossy compression methods such as *JPEG* [4] are efficient for photographic images but do not apply well to palletized images and maps. The lossless *JPEG-LS* [5] uses linear *predictive modeling*, which also works well on natural images where adjacent pixels tend to have similar values. The predictive coding, however, is not efficient for images that contain only few colors.

It is also possible to divide the maps into separate color layers and to apply lossless binary image compression such as *JBIG* and *JBIG2* [6][7]. They use *context-based statistical modeling* and *arithmetic coding* in the same manner as originally proposed in [8]. The probability of each pixel is estimated on the basis of *context*, which is defined as the combination of a set of already processed neighboring pixels. Each context is assigned with its own statistical model that is adaptively updated during the compression process. Decompression is a synchronous process with the compression.

Embedded image-domain adaptive compression of simple images (EIDAC) [9] 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 context pixels are selected not only from the current bit plane but also from the already processed layers.

Another approach which utilizes three-dimensional dependencies is called *SKIP pixel coding* [10]. Binary layers are acquired by color decomposition, and the coding sequence proceeds layer by layer. In a particular layer, if a given pixel has already been coded in a layer of higher priority, it does not need to be coded in the current layer or any of the lower layers. Thus, the coding of large amount of redundant information around blank areas could be “skipped”.

Statistical context-based compression such as the *prediction by partial matching* (PPM) [11] has also been applied to the compression of map images [12]. The method is a 2-D version of the PPM method by combining a 2-D template with the standard PPM coding. The method has been applied to palette images and street maps. A simple scheme for resolution reduction has also been given and the proposed scheme was extended to resolution progressive coding, too.

The *piecewise-constant image model* (PWC) [13] is a technique designed for lossless compression of palette images, which uses two-pass object-based modeling. In the first pass, the boundaries between constant color pieces are established. The color of the pieces are determined and coded in the second pass. The method was reported to give similar to or better compression than the JBIG but with a much faster implementation.

Among the alternative approaches, context-based statistical compression of the color layers is the most efficient in terms of compression. The location of the context pixels can be optimized for each layer separately as proposed in [14]. The method optimizes the location of the template pixels within a limited neighborhood area, and produces the ordered template as the result. The ordering can then be used to derive the context template for any given template size. The method was then applied for generating optimized multi-level context template for map images [15].

Theoretically, better probability estimation can be obtained using a larger context template. The number of contexts, however, grows exponentially with the size of template; adding one more pixel to the template doubles the size of the model. This can lead to *context dilution* problem where the statistics are distributed over too many contexts, thus affecting the accuracy of the probability estimates. The use of *context tree* [16] provides a more efficient approach for the context modeling so that a larger number of neighbor pixels can be taken into account without the context dilution problem.

In this paper, we propose a method for compressing multi-component map images based on layer separation and context tree modeling. The main structure of the method is shown in Figure 1. The paper is based on the ideas and results presented in two recent conference papers: the use of multi-level context tree model as proposed in [17], and the algorithm for obtaining optimal ordering of the layers as proposed in [18]. The acquiring of the optimal layer ordering is related to the directed spanning tree problem, and is solved by an algorithm derived from the Edmond's algorithm for optimum branching. The proposed method gives about 50% better compression than the JBIG.

The rest of the paper is organized as follows. Context-based compression and context tree modeling are recalled in Section 2. Compression of map images is then considered in Section 3 for semantic and color separated layers. Both multi-level template fixed-size template and multi-level context tree models are studied. The problem of obtaining optimal ordering of the layers is studied In Section 4. Experiments are given in Section 5, and conclusions are drawn in Section 6.

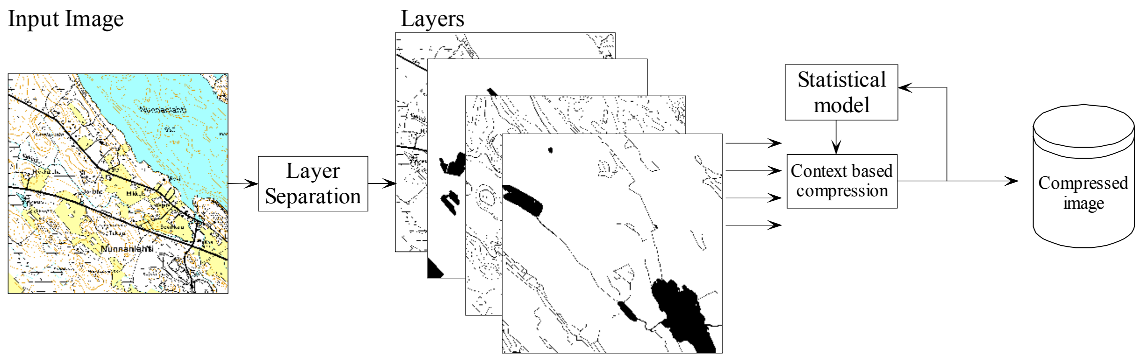


Figure 1: System diagram to illustrate the compression method.

2. Context based compression

Statistical image compression consists of two distinct phases: *statistical modeling* and *coding* [19]. In the modeling phase, we estimate the probability distribution of the symbols to be compressed. The coding process assigns variable length code words to the symbols according to the probability model so that shorter codes are assigned to more probable symbols, and vice versa. The coding can be performed using *arithmetic coding*, which provides optimal coding for the given probability model [20].

2.1. Statistical modeling

Binary image can be considered as a message generated by an information source. The idea of statistical modeling is to describe the message symbols (pixels) according to the probability distribution of the source alphabet (binary alphabet, in our case). Shannon has shown in [21] that the information content of a single symbol (pixel) in the message (image) can be measured by its *entropy*. The entropy of the entire image can be calculated as the average entropy of all pixels:

$$H_{image} = -\frac{1}{n}\sum_{i=1}^{n}\log_2 p_i$$

where p_i is the probability of ith pixel and n is the total number of pixels in the image.

2.2. Context modeling

The pixels in an image form geometrical structures with appropriate spatial dependencies that can be described by *context-based statistical model* [8]. The probability of a pixel is conditioned on a *context C*, which is defined as the black-white configuration of the neighboring pixels within a local template, see Figure 2.

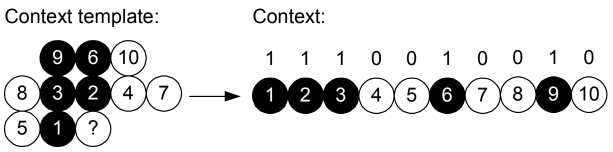


Figure 2: An example of a 10-pixel context.

In principle, better probability estimation can be achieved using larger context template. Large template, however, does not always result in compression improvement because the number of contexts grows exponentially with the size of template; adding one more pixel to the template doubles the size of the model. This leads to the *context dilution* problem, in which the statistics are distributed over too many contexts, and thus, affects the accuracy of the probability estimates.

Nevertheless, only a small fraction of all contexts are really important. For example, in the case of sample binary images with 10-pixel context template, about 50 % of the code bits originate from the few most important contexts as illustrated in Figure 3. Furthermore, 99 % of the code bits originate from 183 contexts, and 429 out of the 1024 contexts are never used at all.

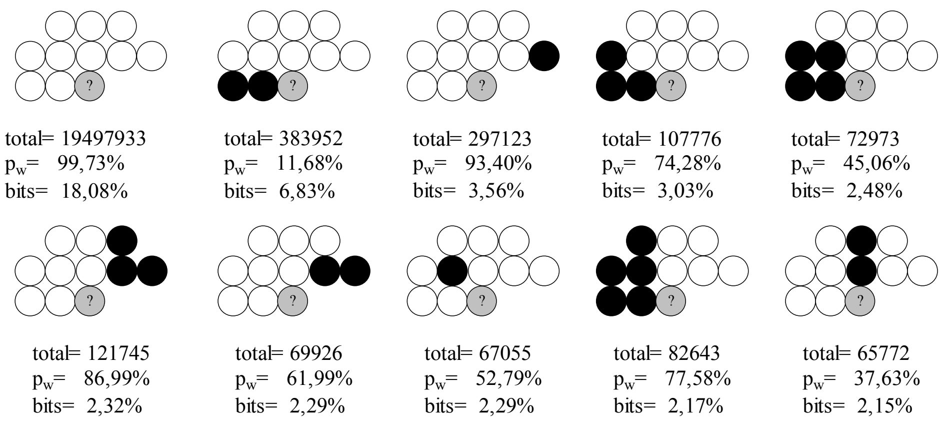


Figure 3: The most important contexts with the 10-pixel template of JBIG.

2.3. Context tree

Context tree provides a more flexible approach for modeling the contexts so that larger number of neighbor pixels can be taken into account without the context dilution problem [16]. The contexts are represented by a *binary tree*, in which the context is constructed pixel by pixel. The context selection is deterministic and only the leaves of the tree are used. The location of the next neighbor pixels and the depth of the individual branches of the tree depend on the combination of the already coded neighbor pixel values. Once the tree has been created, it is fully static and can be used in the compression as any other fixed-size template.

Context tree is applied in the compression in a similar manner as the fixed-size context templates; only the context selection is different. The context selection is made by traversing the context tree from the root to leaf, each time selecting the branch according to the corresponding neighbor pixel value. The leaf has a pointer (index) to the statistical model that is to be used. Each node in the tree represents a single context, as illustrated in Figure 4. The two children of a context correspond to the parent context augmented by one more pixel. The position of this pixel can be fixed in a predefined order, or optimized within a limited search area, relative to the compressed pixel position.

There are different ways for utilizing the context tree. Martins and Forchhammer [16] apply full *k*-level tree that corresponds to a complete collection of all possible contexts up to the order *k*. The context is dynamically chosen along the path from root to leaf using *predictive minimum description length principle* (PMDL) [22]. Only one context is used at a time but statistics are updated for all contexts along the path. The two-level context model of [23] is a special case of this approach. Improvement of about 8 % was reported in [16] but at the cost of high running time because several contexts must be considered for each input pixel to be compressed.

The tree can be optimized beforehand using a training image (*static approach*) [24], or optimized directly to the image to be compressed (*semi-adaptive approach*) [16]. In the latter case, an additional pass over the image is required to collect the statistics, and the tree must also be stored in the compressed file. The cost of storing the tree structure is one bit per node. The static approach is possible because of the similarity of the trees with images of the same type. On the negative side, the resulting tree would be more dependent on the choice of the training image.

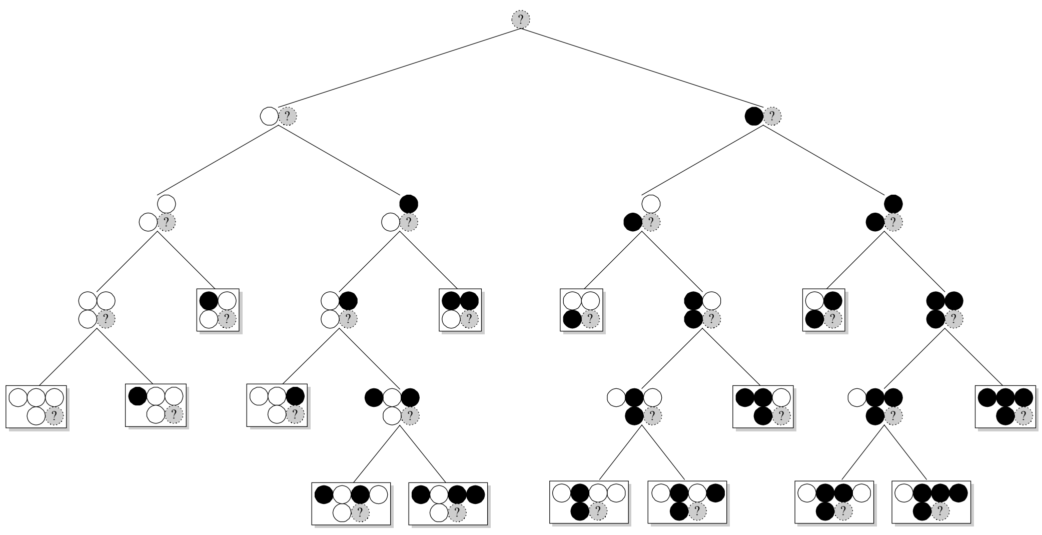


Figure 4: Illustration of a Context tree.

2.4. Construction of the tree

To construct a context tree, the image is processed and statistics are calculated for every context in the full tree including the internal nodes. The tree is then pruned by comparing the parent node and its two sub trees at every level. If compression gain is not achieved by using the two sub trees instead of the parent node, the sub trees are removed and the parent node will become a leaf. The compression gain is calculated as:

$$Gain(C) = l(C) - l(C_{left}) - l(C_{right}) - SplitCost \tag{1}$$

where C is the parent node, and C_{left} and C_{right} are the two sub trees. The code length l denotes the total number of output bits from the pixels coded using the context in the particular node. The cost of storing the tree is integrated into the *SplitCost*. The code length can be calculated by summing up the self-entropies of the pixels as they occur in the image:

$$l(C) = \sum_{t} \log p^{t}(C)$$

The probability of the pixel is calculated on the basis of the observed frequencies using a Bayesian sequential estimator:

$$p(C) = \begin{cases} p_W(C) = \dfrac{n_W(C) + \delta}{n_W(C) + n_B(C) + 2\delta}, & \text{if the pixel is } \textit{white} \\ p_B(C) = 1 - p_W(C), & \text{if the pixel is } \textit{black} \end{cases}$$

where n_W and n_B are the number of times, and p_W and p_B are the probabilities for white and black colors respectively, and $\delta = 0.45$, as in [6].

The construction of the context tree consists of two parts [25]: choosing an order of the context pixels, and constructing optimal context tree for the given order. The following construction strategies have been considered:

- *Optimal bottom-up*: The approach constructs a full tree of k levels, which is then pruned one level at a time using the criterion of (1). The bottom-up approach provides optimal context tree [26] for the given order but at the cost $O(2^k)$ time and space.
- *Optimal top-down*: The tree is constructed by recursively expanding the tree starting from the root using the algorithm in Figure 5. The advantage is that only linear time and space complexity is needed [25].
- *Free-tree*: The tree is constructed stepwise by expanding the tree one level at a time using a local pruning criterion. The location of the context pixel is optimized for each node separately [16]. However, it is possible that the tree terminates too early as in the example in Figure 6. *Delayed pruning* has therefore been proposed to restrain this problem [24].

In a static approach, there is no overhead from storing the tree and the *SplitCost* is 0. In semi-adaptive approach, the cost for storing the tree is 1 bit per node when fixed context order is used. In the case of free-tree, the position of the next context pixel must also be stored for each context. The additional cost of storing the location of this split pixel tends to reduce the overall size of the tree.

```
ConstructOptimalTree(C)
{
    CreateChildrenNodes(C);
    $L_{left} \leftarrow$ ConstructOptimalTree(C.left);
    $L_{right} \leftarrow$ ConstructOptimalTree(C.right);
    $L_{root} \leftarrow H(C)$;
    $Gain \leftarrow L_{root} - L_{left} - L_{right} - SplitCost$;
    IF Gain > 0 THEN
        $L \leftarrow L_{left} + L_{right} + SplitCost$;
    ELSE
        C.left ← NIL;
        C.right ← NIL;
        $L \leftarrow L_{root}$;
    RETURN L;
}
```

Figure 5: Recursive algorithm for optimal construction of the context tree.

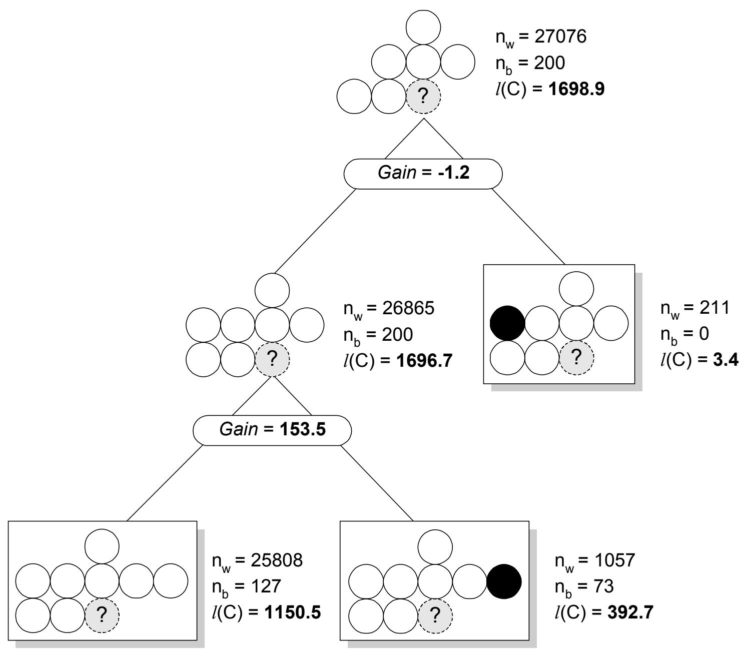


Figure 6: Example of tree pruning with local pruning criterion.

3. Map image compression

Map images can be a result of rasterization of vector map format such as *Simple Vector Format* (SVF), *Scalable Vector Graphics* (SVG) or *ERSI ArcShape* [27]. The map server can provide the maps as a set of layers with different semantic meaning. For example, the topographic map series 1 : 20 000 of *National Land Survey of Finland* (NLS) [28] divides the information into four logical layers: *basic* (topographic data and contours), *elevation lines*, *fields* and *water*. The size of each layer is 5000×5000 pixels, and represents a 10×10 km^2 area. The map image then can be easily reconstructed by combining the binary layers, and displayed to the user as a color image.

3.1. Representation of map image

In order to utilize the context-based compression, the map must be divided into binary layers. Each layer is compressed separately, and the compressed layers are stored into the same file as proposed in [29]. There are three alternative approaches to do this: *semantic separation*, *color separation* and *bit-level separation*.

This first approach, *semantic separation*, means that the map is output into a set of binary layers each containing different semantic meaning. This representation is possible if the maps are obtained directly from a map database. The advantages of semantic separation are better compressibility, and that the layers to be shown can be selected at the time of viewing.

On the other hand, the map image could be provided as a raster color image without any additional information about the semantics of the image. In this case, *color separation* must be used to divide the image into binary layers so that each layer represents one color of the original image. The drawback of the color separation is that information of the original semantic separation cannot be recovered. Furthermore, the color separation can fragment the binary layers as shown in Figure 7. For example, overlapping text elements break the continuation of fields and lakes. This does not decrease the quality of the image but it increases the complexity of these layers, and thus, the compressed file size. Moreover, the color separation results into one additional layer – the background color.

The third approach, *bit-level separation*, must be applied when we have the original map only as a raster image, and the number of colors is too high for efficient color separation. For example, the image might have been digitized from a paper copy and stored using lossy compression method, such as JPEG [4]. In the bit-level separation, the number of colors is first reduced by quantizing the image into a limited size color palette, or to 256 gray-scales. The resulting image is then separated into bit planes using Gray coding [30], and represented as a sequence of binary images. In this paper, we consider only the first two approaches (semantic and color separation).

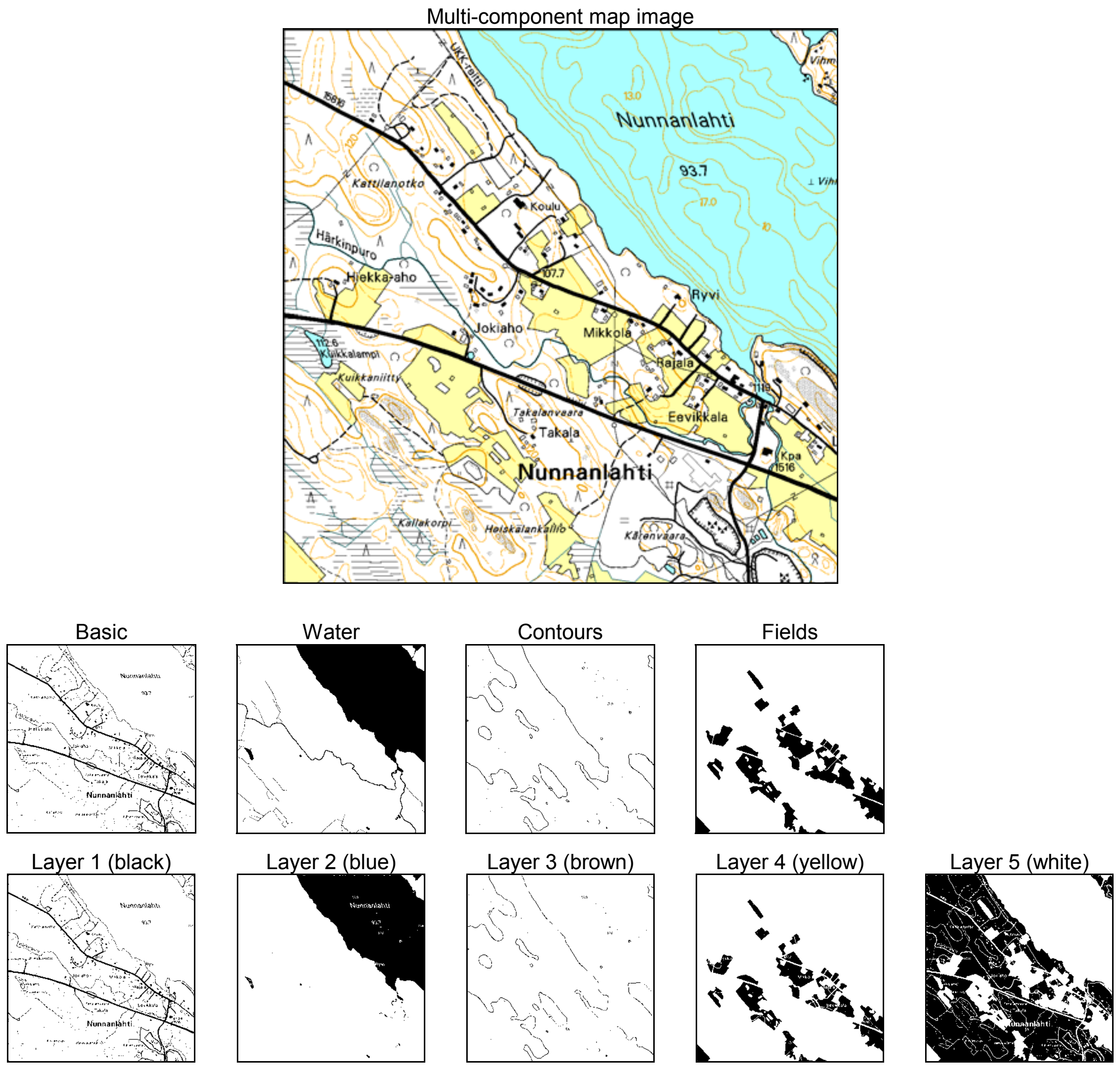


Figure 7: Illustration of a multi-component map image. The shown fragment has the dimensions of 1000 × 1000 pixels.

3.2. Independent compression of layers

Straightforward solution is to compress the layers separately by fixed-size template defined by the standard 1-norm and 2-norm distance functions, see Figure 9. The optimal template size depends on the size of the image. The location of the template pixels, on the other hand, has no direct effect on the learning cost but, if properly designed, may greatly improve the accuracy of the model. It is therefore feasible to optimize the location of the template pixels for the compressed images.

It is possible to optimize the size and shape of the template for the given image layer to be compressed at the cost of longer compression time [14]. The optimal context template can be solved by compressing the image using all possible templates and selecting the one with the most compression. However, there are an exponential number of different template configurations as a function of the template size. A more practical approach is to optimize the location of the template pixels one pixel at a time as outlined in Figure 8.

```
Given:
  Search template T=array[0.. $k_{MAX}$]
  Input image I

Construct_Template(T,I)
{
  while (T != occupied)
  {
    L = FindMinCodeLenghtPosition(T,I);
    AddPositionPermanentely(L,T);
  }
}
```

Figure 8: Algorithm for optimizing the context template.

The optimization starts with an empty context template and expands it by one pixel at a time. A new pixel is added to each unoccupied location in the neighborhood area. For each candidate pixel location, we calculate the statistics of the image and compare the entropies of the model with different location of the candidate pixel. We then select the pixel location providing minimum entropy, and add it permanently to the context template. The process is continued until the context template size reaches a predefined maximum context size k_{max}.

The optimization can be applied in two alternative manners: *static* and *semi-adaptive*. In the static approach, we optimize the template using a priori knowledge of the image type. This is possible, as we know the type of the images to be compressed. The advantage is that the time-consuming optimization can be done off-line. In the semi-adaptive approach, the template is optimized for the compressed image and is stored in the compressed file. This is a better solution if the image type is not known beforehand. The compression process, however, would be very slow, which makes this approach not suitable for real-time applications.

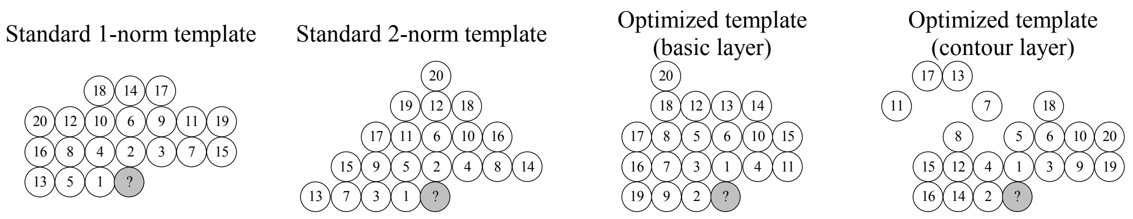

Standard 1-norm template
Standard 2-norm template
Optimized template (basic layer)
Optimized template (contour layer)
?

Figure 9: Alternative orderings for the context templates.

3.3. Optimized multi-level template

The idea of *multi-layer* context template is to utilize the information from additional image layer, referred here as the *reference image*. The restriction on the use of the reference image is that it must have already been coded so that both encoder and decoder have the same information. The main difference in the construction of *single-layer* and *multi-layer* context templates is in additional neighborhood mask used for selection of the pixels from the reference image. The pixels in the current layer must be already coded pixels, but in the reference layer the pixels can be anywhere in the image.

The idea of utilizing inter-layer dependencies have been used in JBIG2 [7] where pattern matching technique is applied to enhance the compression of document images. The image is divided into separate segments (typically text symbols), which are each coded as an index to a similar segment in a dictionary, and the compression of the pixels in the segment. Two-layer context template is applied for utilizing the information of the matching of the symbol when compressing the pixels in the segment. The same idea was then applied for generating multi-level context template for map images [15].

In [31] gray-scale images are divided into binary layers by Gray coding and bit plane separation. The bit planes are then compressed as binary images using context template including pixels both from the current level, and from the pixels in the previously coded (higher level) bit planes.

In the following, we use the combined neighborhood area of 77 pixels as shown in Figure 10. Optimized sample context templates for the map images are shown in Figure 11. The two-level context template of JBIG2 is shown for comparison.

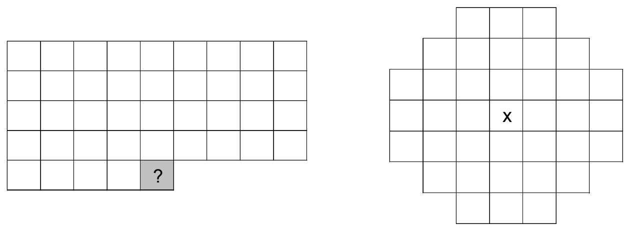


Figure 10: The neighborhood area for optimizing the location of the template pixels. The area in the current layer is shown left, and in the reference layer right.

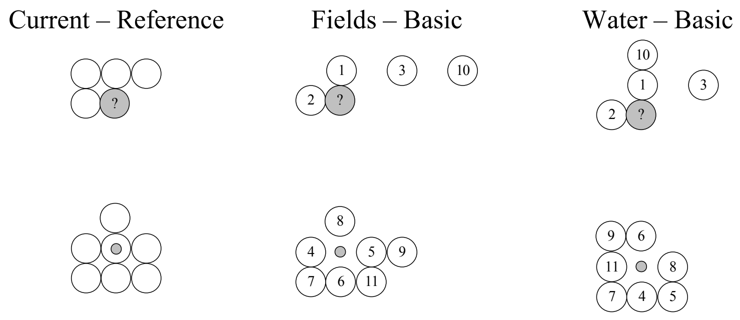


Figure 11: Sample two-level context templates for JBIG-2 (left), optimized for *fields* layer when using (middle), and optimized for *water* layer (right) when the *basic* layer is used as the reference layer.

3.4. Multi-level context tree

The idea of utilizing multi-layer dependencies can be extended also to the context tree modeling. The multi-level context tree is constructed as follows. The tree starts from scratch and the branches are expanded one pixel at a time. The location of the template pixels are optimized and fixed beforehand and then applied for every branch. Another approach is to optimize the location separately for every branch (Free Tree approach). The context pixels are chosen from the same joint 77 pixels neighborhood as shown in Figure 10.

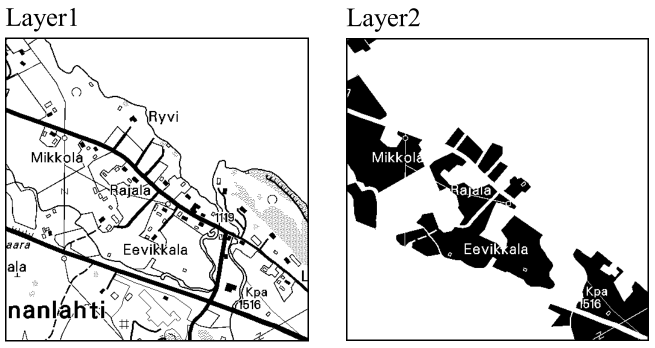


Figure 12: Example of two layers obtained by color separation.

The use of the information from the reference layer will allow us in some cases to increase the compression ratio of the single layer up to 50% according to [17]. In fact, if we will consider the compression of sample images in Figure 12, the compression of these two images separately using single-level context trees would result in 4854+1330=6184 bytes. On the other hand, if we use the information from the first layer when compressing the second layer, the tree structure of the second layer would be simpler. All information would be concentrated only in the first branch of the tree, as shown in Figure 13. Thus the compression of the second layer would be only 146 bytes, and the final size of the compressed file 4854+146=5000 bytes.

The map images usually have inter-layer dependencies. For example, the same pixel is usually not set in the *water* layer and in the *field* layers at the same time although it is possible as the layers are generated from map database independently from each other. Another observation is that the *basic* and the *water* layers have redundant information along the rivers and lake boundaries. In general, anything is possible, and it is not easy to observe the existing dependencies by the eye. The dependencies, however, can be automatically captured by the statistical modeling.

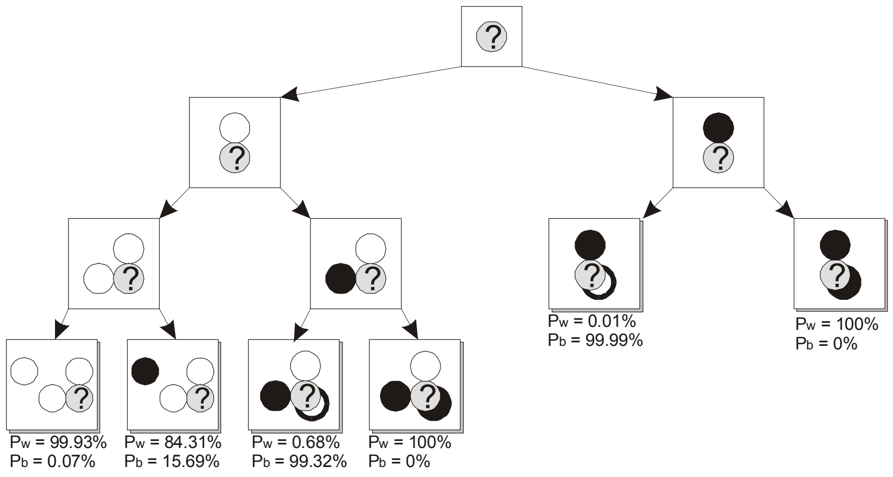


Figure 13: Example of a two-level context tree, in which two context pixels are taken from the current layer and one from the reference layer (shown below the current pixel).

4. Optimal ordering of the layers

The existing dependencies are demonstrated in Figure 14, in the case of the NLS map images. There are significant inter-layer dependencies between the *basic* layer and the two other layers (*water*, *field*). The *contour* layer, on the other hand, is independent from all other layers. The main observation is that we cannot utilize all the existing dependencies as the order of processing restricts which layers we can use as the reference layer.

For example, if we compress the basic layer first, we can then improve the compression of the water layer by 52% (118705 bytes). The opposite order would improve the compression of the basic layer by 35% (345061 bytes). It is easy to see that the best order of these layers would be to compress first the water layer, second the basic layer, and then the fields layer last. The contours layer should be processed independently.

In general, we can select any predefined order on the basis of known (or assumed) dependencies. If the image source is not known beforehand, we should find the optimal order of the layers for maximal utilization of the inter-layer dependencies. The selected processing order can be saved in the compressed file using a few bits. The problem of finding the optimal order is studied in the following subsections.

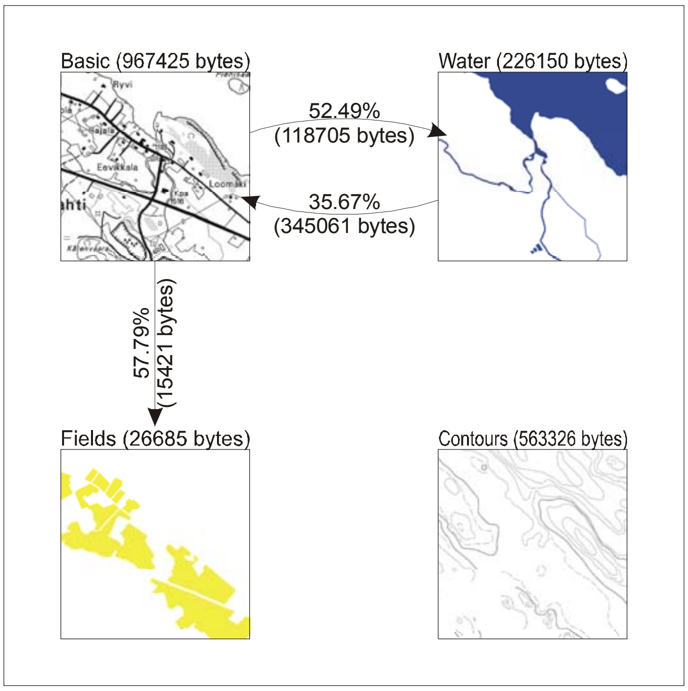


Figure 14: The arrows show the inter-layer dependencies as the number of saved bits when compressing the second image using the first one as reference image.

4.1. Construction of the cost matrix

Suppose that we have *k* layers. In order to determine the optimal layer ordering we have to consider all pairwise dependencies by tentatively compressing every layer using all other layers as reference. This results into a *k*×*k cost matrix* consisting of the absolute bit rates for every *layer-reference layer* pairing, see Table 1. On the basis of the cost matrix, we can generate all *k*! possible permutations for the processing order and calculate the total saving achieved by the given layer ordering. If the number of layers is small enough (with the NLS images *k*=4), this is not a problem. With larger values of *k*, however, this could be computationally too expensive.

On the other hand, not all information in the matrix is relevant to us. In the case when there are no dependencies between the layers, the corresponding compression result would be the same (or worse) with or without the use of inter-layer context model. We can therefore reduce the amount of information in the cost matrix by subtracting the original values (Table 2) by the values obtained by layer-independent compression (Table 1), and eliminate values smaller than or equal to zero. The resulting cost matrix is shown in Table 3.

The reduced cost matrix can be considered as a directed graph with *k* nodes as shown in Figure 15. The problem of finding the optimal order is closely related (but not exactly) to the *minimum spanning tree* problem. We will next give graph-based algorithm to solve the optimal ordering. We follow the approach taken by Tate for optimal band ordering in the compression of multi-spectral images [32].

Table 1: Example of the cost matrix for a sample image.

	0	1	2	3	4	5	6	7	8	9	10	11
0		7564	13013	6189	1792	10448	16472	28477	1619	13683	13360	23976
1	17158		16041	5871	1731	9997	15713	27988	1557	11994	11377	22000
2	13488	8594		6029	1834	10461	16488	27318	1946	13671	13105	23760
3	23390	12116	20700		1636	9931	12333	28093	2055	13567	12735	22189
4	23471	12789	21348	6091		11011	18168	30445	2239	13683	12682	23713
5	23293	12116	21144	5385	1751		15506	26732	1834	13214	12102	22548
6	23536	12269	21423	3934	1681	10134		27405	2111	13674	13385	20479
7	20197	10149	17117	4889	1646	7007	12446		1737	11075	10612	13574
8	22810	11956	22493	6780	1834	10617	16899	29489		13256	13097	23976
9	26015	12029	20013	6467	1834	10329	16494	29514	2239		9401	22764
10	23036	11617	19305	5628	1722	10154	15797	27815	2013	7344		22038
11	26169	12561	21008	5933	1829	10660	14706	23261	1963	12231	12515	

Table 2: The original compression values obtained without the use of dependencies.

0	1	2	3	4	5	6	7	8	9	10	11
23451	11847	20123	5504	1734	10256	15341	29206	1845	12328	11495	23049

Table 3: Example of the cost matrix after reduction.

	0	1	2	3	4	5	6	7	8	9	10	11
0		4283	7110	0	0	0	0	729	226	0	0	0
1	6293		4082	0	3	259	0	1218	288	334	118	1049
2	9963	3253		0	0	0	0	1888	0	0	0	0
3	61	0	0		98	325	3008	1113	0	0	0	860
4	0	0	0	0		0	0	0	0	0	0	0
5	158	0	0	119	0		0	2474	11	0	0	501
6	0	0	0	1570	53	122		1801	0	0	0	2570
7	3254	1698	3006	615	88	3249	2895		108	1253	883	9475
8	641	0	0	0	0	0	0	0		0	0	0
9	0	0	110	0	0	0	0	0	0		2094	285
10	415	230	818	0	12	102	0	1391	0	4984		1011
11	0	0	0	0	0	0	635	5945	0	97	0	

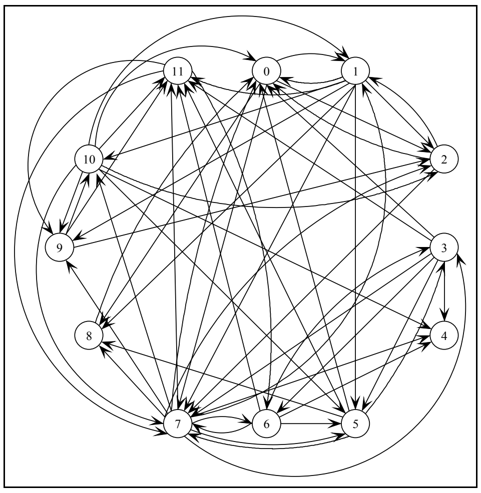


Figure 15: Graph representation of Table 3

```
Given:
  Connected graph G=[V,E]
  Solution set S=[V,E]

MST_For_Directed_Graph(G,S)

  FOR (each Vi) DO
    Ei = FindMaxEnteringEdge(Vi,G);
    AddEdgeAndItsEndpoints(S,Ei);

  C=LocateCycles(S);
  IF (C != empty) THEN
    FOR (each Ci)
      Ee=FindEnteringEdges(G,Ci);
      CalculateModifiedCost(Ee);
      Eem=FindMaxEdge(Ee);
      ReplaceEdge(Ec,Eem);
```

Figure 16: Edmond's algorithm.

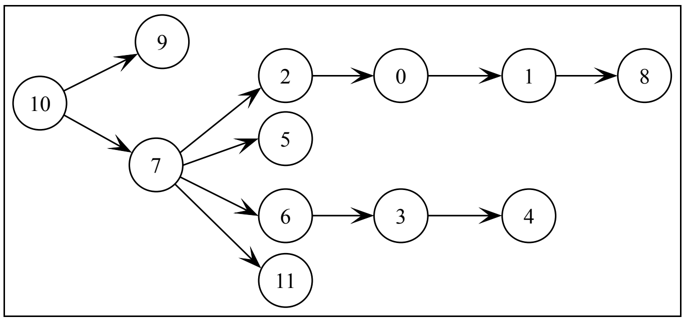


Figure 17: The resulting spanning tree after Edmonds' algorithm.

4.2. Maximum spanning tree

A spanning tree of a graph is a subset of the edges so that all nodes of the graph are included in the set but there are no cycles. *Minimum spanning tree* (MST) is a spanning tree with the minimum sum of the weights of the edges included in the given graph [33].

The minimum weighted tree can be solved in polynomial time using *Prim's algorithm* [34], for example. The algorithm begins by adding the lowest weighted edge into the solution. It then loops by adding the next lowest weighted edge that connects a node inside the set to another node outside of the set. The algorithm terminates when n-1 edges have been chosen and, thus, all nodes covered.

However, there are few differences that separate our problem of obtaining the optimal order from the minimum spanning tree problem:

- We have a directed graph whereas the MST is defined with undirected graph.
- We can have only one incoming edge for any node.
- We can have several separate spanning trees instead of only one.
- We have maximization problem.

The first two differences make the problem as a *directed spanning tree problem*. The directed spanning tree is defined as a spanning tree where all nodes (except the root) have exactly one incoming edge. This is also known as the *optimum branching* problem [35], and can be solved in $O(n^2)$ time [36].

In the optimal ordering, it is not necessary to have a single spanning tree but we can have separate sub graphs, see Figure 14. This means that we should actually find spanning forest instead of a single tree. The problem was considered as the *maximum spanning forest* problem in [32]. However, we have eliminated all negative weights in the cost matrix (Table 3), and the inclusion of a zero-edge can be considered as independent compression of the corresponding layers. Thus, we can still consider the optimal ordering as directed spanning tree problem.

We apply Edmond's algorithm [35] as shown in Figure 16. The algorithm begins by selecting the maximum incoming edge for every node except the root. If no cycles are formed the resulting graph is MST. Otherwise, the algorithm detects and removes existing cycles by removing an edge in the cycle and replacing it by another edge incoming from outside. The edges are chosen so that maximal saving is achieved. We also note that the Edmond's algorithm requires that one node is selected as the root. We therefore repeat the algorithm using every node as potential root and select the one resulting to most saving. The time complexity of the algorithm for single root is $O(n^2)$, and for considering all nodes as a root $O(n^3)$.

The optimal branching for the data in Table 1 is shown in Figure 17. This ordering of the layers sums up to 124977 bytes, which corresponds to the improvement of 24.79 % in comparison to the original result. The process of the algorithm in detail is illustrated in Figure 18 using node 10 as the root. The initial solution contains three cycles (0↔2, 3↔6, 7↔11). The first cycle (0↔2) is eliminated by replacing the edge 0→2 by the edge 1→2. This creates a new cycle (0→1→2), which is then resolved in the second step by replacing the edge 1→2 by the edge 7→2. The rest of the cycles are resolved accordingly.

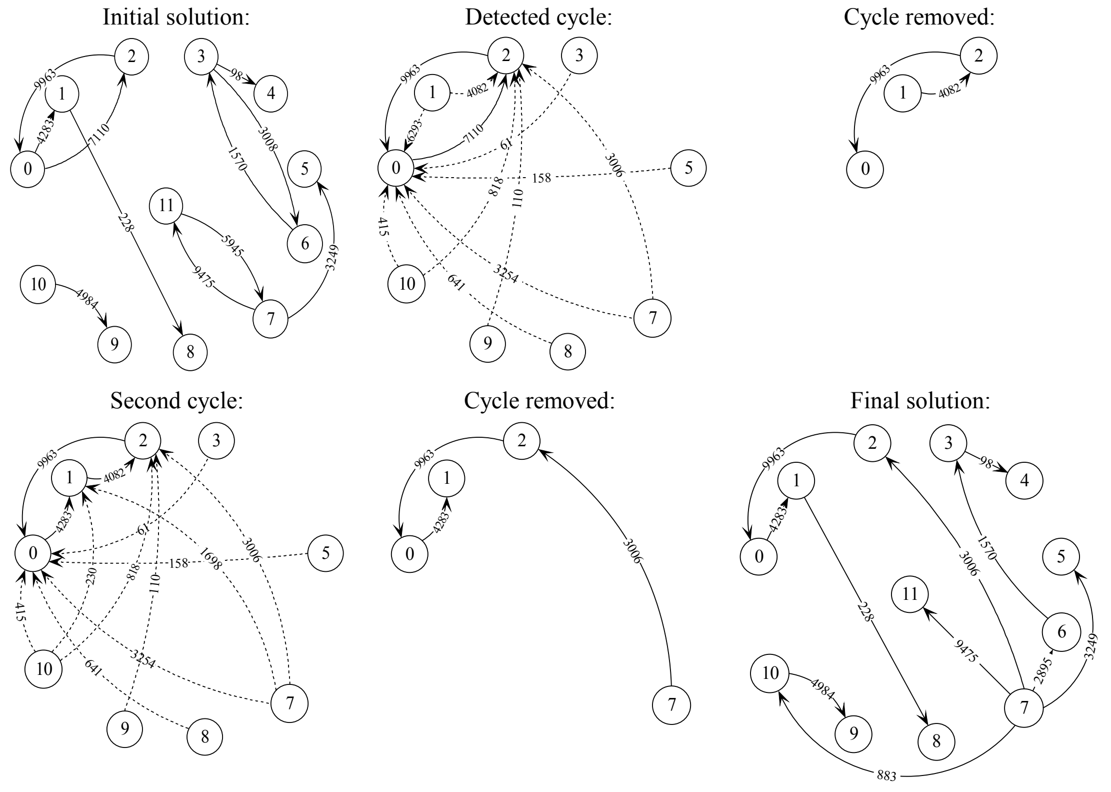


Figure 18: The process of Edmond's algorithm. The first graph is the result of selecting maximum incoming edge except the root (initially node 10). The following steps illustrate the removal of cycles, and the final result.

4.3. Selection of the background color

In the case of color separation, we can eliminate one layer completely and consider it as the *background color*. The background color is usually white but this is not necessarily the case always. In fact, we can set any layer as the background color. The advantage is that the chosen layer is not stored in the compressed file at all. There are two obvious choices for selecting the background color:

- *Greedy*: The layer with the maximal compressed size.
- *Optimal*: The layer of whose removal gives most improvement in compression.

The greedy choice is not necessary the best choice because the layer with most information can also include most inter-layer dependencies with other layers. In other words, the background layer cannot be used as a reference layer, and therefore, the removal of the dependent layer can increase the compressed size of other layers.

The optimal choice can be obtained by considering all layers as the background color, and selecting the one that result in the best overall compression. In principle, this is computationally demanding as the problem of finding the optimal ordering takes $O(n^3)$, and thus, $O(n^4)$ time for finding the optimal choice for the background color. In practice, however, the number of layers is small (here n=4..16). Therefore, the bottleneck of the optimization is not the solving of the spanning tree but the calculation of the cost matrix at the first place.

5. Experiments

We evaluate the proposed method by compressing the sets of map images listed in Table 4, and illustrated in Figure 19. The sets #1 to #4 are from the map database of the *National Land Survey of Finland* (NLS) [28]. In the case of the detailed road maps (*Sea* and *City*), the number of colors was reduced to 16 before the compression.

Two different image representations are considered: (1) semantic representation and (2) color separation. Semantic representation was available for four randomly selected map sheets of scale 1 : 20 000. These images are further denoted as 124101, 201401, 263112 and 431204 according to their map sheet number in the NLS database. The rest of the images were passed trough color separation. The obtained binary layers (semantic or color separation) were compressed using context tree modeling and arithmetic coding. The optimal ordering and the choice of background color were solved for every image separately.

Table 4: Set of test images.

	Scale	Type	Images	Image size	No. of colors
Semantic:	1 : 20 000	Topographic	4	5000×5000	5
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	Road map	4	1024×1024	16
Set #4:	1 : 800 000	Road map	4	1024×1024	16
Set #5:	1 : 250 000	Road map (detailed)	2	800×800	16

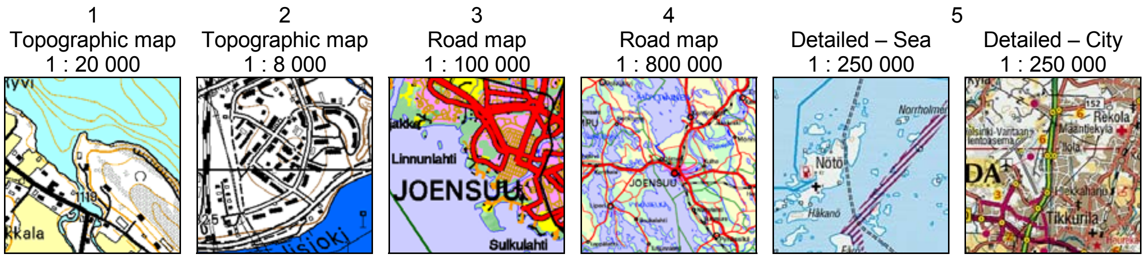


Figure 19: Sample 256×256 pixel fragments of the test images.

We first study the effect of the multi-layer context tree modeling by compressing the set of images by the following variants:

- CT: Context tree modeling of each layer separately.
- MCT (single): Multi-level context tree using single conditioning layer.
- MCT (optimized): Multi-level context tree using optimal layer ordering.

All variants are considered with and without optimal selection of the background color. In the semantic separation the background is excluded by default as it was never included. The results show that the multi-level context tree (MCT) produces file sizes of about 24% less than the single level context tree (CT), see Table 5. The optimal ordering is about 12% better than the use of a single conditioning layer.

The compression performance is illustrated as a function of the number of contexts in Figure 20, and compared to the fixed size context modeling (JBIG), single-level context tree (CT), and fixed multi-level context (Fixed multi-layer). In this experiment, the number of contexts was explicitly forced to given value for the sake of comparison. The results indicate that better compression can be obtained by increasing the context size (JBIG curve), by using context tree modeling (CT vs. JBIG), and by using multi-level contexts (Fixed multi-layer vs. JBIG). Nevertheless, the joint application of them with the optimized layer ordering gives significantly better compression than any of the ideas alone.

Table 5: Average compression results (kilobytes) for the set of test images.

Semantic separation	With background			Background removed		
	CT	MCT single	MCT optimized	CT	MCT single	MCT optimized
124101	314	218	189	205	174	162
201401	1531	1064	965	1159	823	727
263112	556	317	316	310	270	184
431204	1207	715	707	712	592	590
431306	1192	744	705	696	617	343
Average	**960**	**612**	**577**	**616**	**495**	**401**

Color separation	With background			Background removed		
	CT	MCT single	MCT optimized	CT	MCT single	MCT optimized
1:20000	960	612	577	616	495	401
1:8000	64	37	36	34	32	30
1:100000	243	211	194	185	173	155
1:800000	268	240	219	240	217	198
Sea	181	134	124	125	113	106
Vantaa	215	196	186	190	173	164
Average	**322**	**238**	**223**	**232**	**201**	**176**

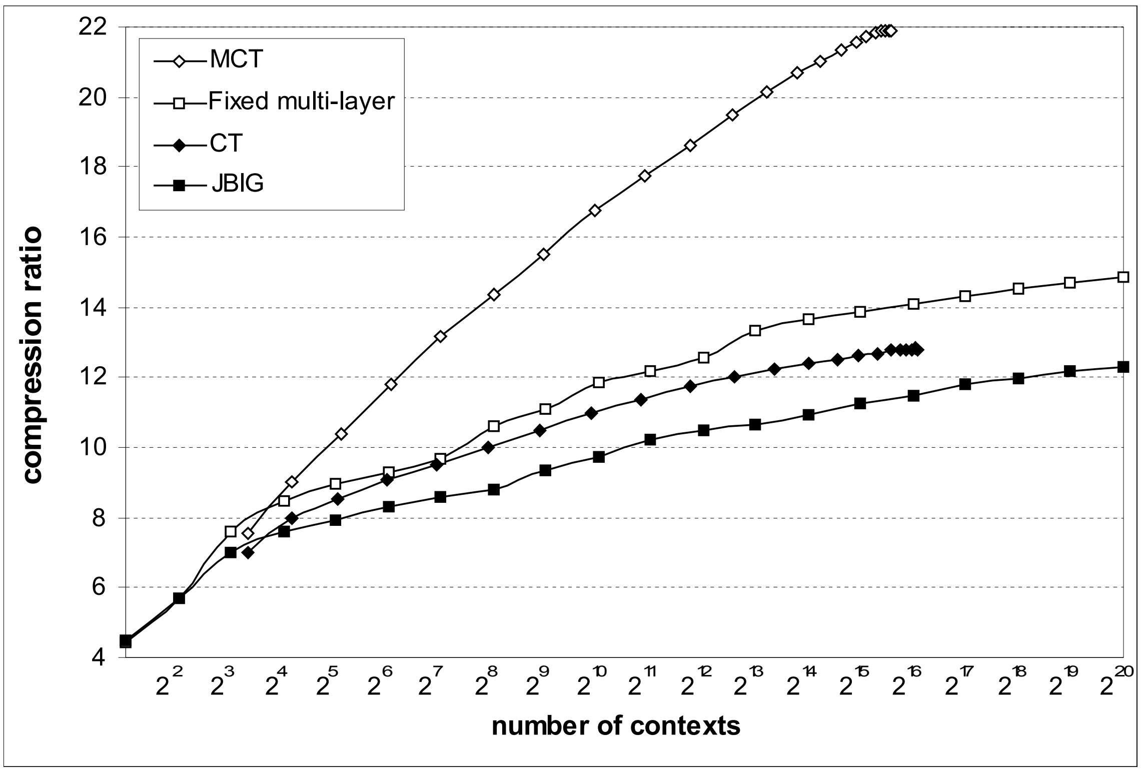


Figure 20: Compression result (for semantic decomposition) as the function of the number of contexts.

The improved compression, however, does not come without a price. In this case, the drawback is a significantly increased compression time, see Table 6. Most of the time is spent for calculating the cost matrix, i.e. tentatively compressing the image layers using all possible *current-reference* layer pairs. Finding the optimal ordering does not take much time at all although the algorithm itself has the time complexity of $O(n^4)$. The actual compression is somewhat slower than that of the CT but the difference is tolerable. In comparison to the JBIG, both the CT and MCT are somewhat slower also in the decompression.

Table 6: Compression and decompression times using 500 MHz Pentium III.

	JBIG		CT		MCT			
	Comp.	Decomp.	Tree + Comp	Decomp.	Cost matrix	Optimal ordering	Comp.	Decomp.
Semantic	0:02:23	0:02:09	0:45:02	0:05:38	61:34:00	0:00:04	0:03:10	0:05:39
Set #1	0:03:44	0:03:37	1:39:27	0:11:05	111:59:00	0:00:05	0:06:23	0:11:05
Set #2	0:00:09	0:00:10	0:02:35	0:00:18	2:46:00	0:00:04	0:00:10	0:00:18
Set #3	0:00:22	0:00:23	0:08:58	0:01:03	28:16:00	0:00:04	0:00:45	0:01:03
Set #4	0:00:22	0:00:24	0:09:23	0:01:07	33:02:00	0:00:04	0:00:38	0:01:07
Set #5	0:00:08	0:00:08	0:02:51	0:00:21	22:48:00	0:00:02	0:00:12	0:00:21
Sum	**0:07:06**	**0:06:50**	**2:45:28**	**0:19:29**	**260:25:00**	**0:00:23**	**0:11:18**	**0:19:31**

For illustrating the performance of the CT and MCT in detail, we have collected four most used contexts and their statistics in Figure 21. In the case of independent compression, the optimized context trees can be remarkably different for different layers. For example, the contexts in *Fields* layer contain much fewer pixels than the contexts in *Contours* layer. Pixels are also taken further away and only a few nearby pixels are needed to model the probability distribution. The *Contours* layer does not contain large convex shapes as the *Fields* layers but more complex thin lines that need more context pixels to be modeled accurately.

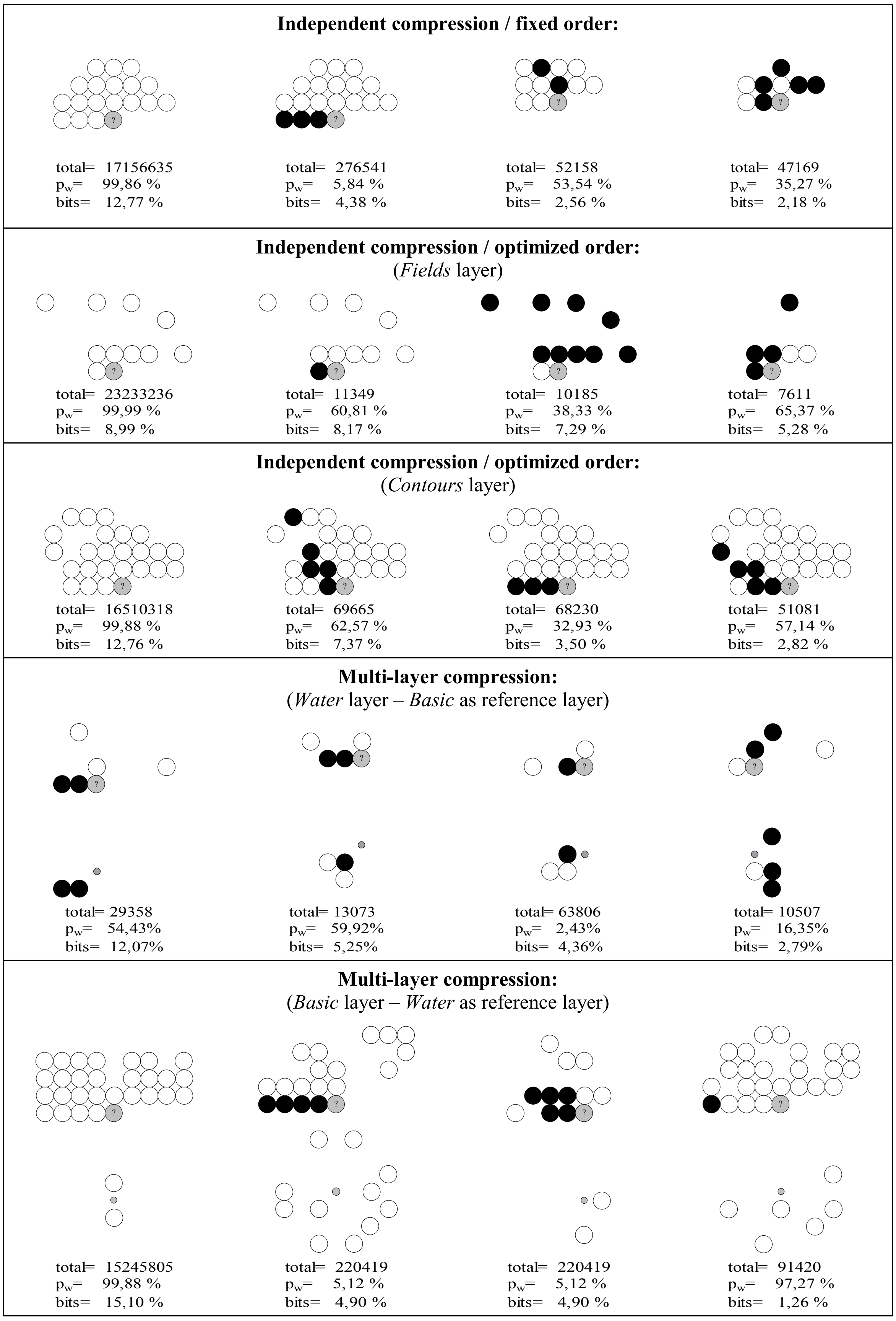
In the case of multi-layer context tree, the selection of the context pixels depends on the layers in question. The general tendency is the same as in the case of independent compression so that *Fields* and *Water* layers need fewer context pixels. The *Basic* layer uses significantly more pixels although the number of pixels depends on the particular pixel combination. The number of the context pixels in the reference layer is typically much smaller. When *Basic* layer is used as reference, only a few nearby pixels are used. In the case of *Water*, on the other hand, the context pixels are scattered from wider area.

Finally, the result of the proposed method (MCT) is compared with the following methods:

- GIF: CompuServe graphics interchange format [1],
- PNG: portable network graphics [2][3],
- JBIG: Joint bi-level image group [6],
- PPM: prediction by partial matching [11][12],
- PWC: Piecewise-constant image model [13],
- SKIP: Explicit skipping of pixels set in previous layer [10],
- CT: Single level context tree [16][24].

The results are summarized in Table 7 (semantic layers) and Table 8 (color separation). Note that the results of GIF, PNG, PPM and PWC are applied to the color images, and the JBIG, SKIP, CT and MCT for the binary layers. Note also that SKIP applies only to color separation where the exclusion of certain pixels can be concluded from the existence of the pixel in another layer. In semantic separation, however, the same pixel may be set into several layers although only one color is printed in the output image.

The result show that the MCT is gives the best compression with all test sets. In the case of color separation, the SKIP method is rather effective in comparison to JBIG despite of its simplicity. Also PWC and CT perform rather well. In the case of semantic separation, the binary image coding methods (JBIG, CT, MCT) are clearly better than the others, of which the MCT gives most compression.



Independent compression / fixed order:

total= 17156635	total= 276541	total= 52158	total= 47169
p_w= 99,86 %	p_w= 5,84 %	p_w= 53,54 %	p_w= 35,27 %
bits= 12,77 %	bits= 4,38 %	bits= 2,56 %	bits= 2,18 %

Independent compression / optimized order:
(*Fields* layer)

total= 23233236	total= 11349	total= 10185	total= 7611
p_w= 99,99 %	p_w= 60,81 %	p_w= 38,33 %	p_w= 65,37 %
bits= 8,99 %	bits= 8,17 %	bits= 7,29 %	bits= 5,28 %

Independent compression / optimized order:
(*Contours* layer)

total= 16510318	total= 69665	total= 68230	total= 51081
p_w= 99,88 %	p_w= 62,57 %	p_w= 32,93 %	p_w= 57,14 %
bits= 12,76 %	bits= 7,37 %	bits= 3,50 %	bits= 2,82 %

Multi-layer compression:
(*Water* layer – *Basic* as reference layer)

total= 29358	total= 13073	total= 63806	total= 10507
p_w= 54,43%	p_w= 59,92%	p_w= 2,43%	p_w= 16,35%
bits= 12,07%	bits= 5,25%	bits= 4,36%	bits= 2,79%

Multi-layer compression:
(*Basic* layer – *Water* as reference layer)

total= 15245805	total= 220419	total= 220419	total= 91420
p_w= 99,88 %	p_w= 5,12 %	p_w= 5,12 %	p_w= 97,27 %
bits= 15,10 %	bits= 4,90 %	bits= 4,90 %	bits= 1,26 %

Figure 21: Most used contexts in various context tree modeling strategies.

Table 7: Compression results (kilobytes) for the methods in comparison for semantic separation.

Semantic	GIF	PNG	JBIG	PPM	PWC	SKIP	CT	MCT
124101	622	962	145	370	153	---	130	106
201401	2633	3599	702	1846	829	---	611	528
263112	1170	1691	277	801	320	---	246	223
431204	2567	3648	618	1889	707	---	544	443
Average	**1748**	**2475**	**435**	**1226**	**502**	**---**	**383**	**325**

Table 8: Compression results (kilobytes) for the methods in comparison for color separation.

Color separation	GIF	PNG	JBIG	PPM	PWC	SKIP	CT	MCT
1:20000	1801	1854	1017	1449	777	532	960	401
1:8000	86	90	62	81	35	34	64	30
1:100000	288	278	283	203	198	202	243	155
1:800000	303	287	274	211	197	198	268	198
Sea	150	155	181	113	124	117	181	106
Vantaa	225	212	238	172	155	172	215	164
Average	**475**	**479**	**343**	**372**	**247**	**209**	**322**	**176**

6. Conclusions

We have proposed a method for compressing map images by multi-level context tree modeling and by optimizing the order of the processing of the binary layers. Solutions are given for the context modeling, utilization of the multi-level dependencies, and for the optimal ordering of the layers. The proposed method gives about 30 % improvement over previous methods on average, in comparison to the compression of binary layers without utilizing the inter-layer dependencies. The optimal order of processing the layers was considered as directed spanning tree problem and solved by an algorithm derived from the Edmond's algorithm.

There are ideas not dealt in this paper that could be considered in the future. For example, the two-level context modeling could be generalized to 3-D by conditioning the probability of the pixel on several layers instead of only one. However, it is not obvious how the optimal order of layers should be solved, and whether this modification would result additional compression worth the trouble. Another idea would be to model the image as a color image instead of dividing it into binary layers. In this case, the context modeling would become significantly more complex. A third idea worth further studies might be to develop the method towards real-time compression by designing faster heuristic for estimating the inter-layer dependencies instead of the brute force approach.

References

[1] T.A. Welch, "A Technique for High-performance Data Compression," *Computer* 17(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 and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, 23(3) pp. 337-343, May 1977.

[4] W.B. Pennebaker, J.L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, 1993.

[5] M. Weinberger, G. Seroussi, G. Sapiro, and M. W. Marcellin, "The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS," *HPL98-193*, HP Labs, 1998.

[6] JBIG: ISO/IEC International Standard 11544, *ISO/IEC/JTC1/SC29/WG9;* also ITU-T Recommendation T.82. Progressive Bi-level Image Compression, 1993.

[7] P.G. Howard, F. Kossentini, B. Martins, S. Forchammer and W.J. Rucklidge, "The emerging JBIG2 standard", *IEEE Trans. on Circuits and Systems for Video Technology*, 8 (7), 1998, 838-848, 1998.

[8] G.G. Langdon, J. Rissanen, "Compression of black-white images with arithmetic coding", *IEEE Trans. Communications* 29 (6): 858-867, 1981.

[9] Y. Yoo, Y. Kwon and A. Ortega, "Embedded Image-Domain Adaptive Compression of Simple Images," *32nd Asilomar Conf. on Signals, Systems and Computers, Nov. 1998.*

[10] S. Forchhammer and O.R. Jensen, "Content layer progressive coding of digital maps", *IEEE Trans. on Image Processing* 11 (12): 1349-1356, December 2002.

[11] J.G. Cleary and I.H. Witten, "Data compression using adaptive coding and partial string matching", *IEEE Trans. on Communications*, 32(4): 396-402, 1984.

[12] S. Forchhammer and J.M. Salinas, "Progressive coding of palette images and digital maps", *IEEE Proceedings Data Compression Conference*, Snowbird, Utah, USA, pp. 362-371, 2002.

[13] P.J. Ausbeck Jr., "The piecewise-constant image model", *Proceedings of the IEEE*, 88 (11), 1779-1789, 2000

[14] E.I. Ageenko, P. Kopylov and P. Fränti "Optimizing context template for compression of multi-component map images", *GraphiCon'00*, Moscow, Russia, 151-156, 2000.

[15] E.I. Ageenko, P. Kopylov and P. Fränti, "On the size and shape of multi-level context templates for compression of map images", *IEEE Int. Conf. on Image Processing (ICIP'01)*, Thessaloniki, Greece, pp. 458-461, vol.3, October 2001.

[16] B. Martins, S. Forchhammer, "Bi-level image compression with tree coding", *IEEE Trans. Image Processing* 7 (4): 517-528, 1998.

[17] P. Kopylov and P. Fränti, "Context tree compression of multi-component map images", *IEEE Proc. Data Compression Conference*, Snowbird, Utah, USA, pp. 212-221, 2002.

[18] P. Kopylov, and P. Fränti, "Optimal layer ordering in the compression of map images", *IEEE Proc. Data Compression Conference* (*DCC'03*), Snowbird, Utah, USA, 323-332, April 2003.

[19] J.J. Rissanen and G.G. Langdon, “Universal modeling and coding”, *IEEE Trans. on Information Theory* IT-27: 12-23, 1981.

[20] J.J. Rissanen and G.G. Langdon, “Arithmetic coding”, *IBM Journal of Research, Development* 23: 146-162, 1979.

[21] C.E. Shanon, “A mathematical theory of communication”, *Bell Syst. Tech Journal* 27: 398-403, 1948.

[22] J. Rissanen, *Stochastic Complexity in Statistical Inquiry*. World Scientific, 1989.

[23] A. Moffat, “Two-level context based compression of binary images”, *IEEE Proceedings Data Compression Conference*, Snowbird, Utah, 382-391, 1991.

[24] P. Fränti and E.I. Ageenko, “On the use of Context tree for binary image compression”, *Proc. IEEE International Conference on Image Processing (ICIP’99)*, Kobe, Japan, vol. 3, pp. 752-756, 1999.

[25] H. Helfgott and M. Cohn, “Linear-time construction of optimal context trees”, *IEEE Proceedings Data Compression Conference*, Snowbird, Utah, 369-377, 1998.

[26] R.I. Nohre, “*Topics in Descriptive Complexity”,* PhD Thesis, University of Lingköping, Sweden, 1994.

[27] ESRI, “ESRI Shapefile Technical Description”, An ESRI White Paper, 1998. (http://www.esri.com/library/whitepages/pdfs/shapefile.pdf)

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

[29] Ageenko E.I., Fränti P., “Compression of large binary images in digital spatial libraries”, *Computers & Graphics* 24 (1): 91-98, February 2000.

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

[31] M. Rabbani and P.W. Melnychuck, “Conditioning contexts for the arithmetic coding of bit planes”, *IEEE Trans. on Signal Processing*, 40 (11): 232-236, 1992.

[32] S.R.Tate, “Band ordering in lossless compression of multispectral images”, *IEEE Trans. on Computers*, 46(4): 477-483, April 1997.

[33] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1990.

[34] R.C. Prim, “Shortest connection networks and some generalizations”, *Bell Systems Technology Journal,* 36: 1389-1401, 1957.

[35] J. Edmonds, “Optimum branchings”, *J. Research of the National Bureau of Standards*, 71B: 133-240, 1967.

[36] R.E. Tarjan, “Finding Optimum Branchings”, *Networks*, 7: 25-35, 1977.