Semi-adaptive Dictionary Based Compression of Map Images

Alexandre Akimov, Pavel Kopylov and Pasi Fränti
University of Joensuu
Joensuu, Finland

Abstract

One of the modern fields of application of the image compression is personal navigation, where the user needs maps in real time. Usually the mobile communication devices do not have the same computational power as the workstations (small memory, narrow communication channel). Thus, the map images must be compressed before transmitting. At the same time, decoding of compressed images must be quick. The existing dictionary based image compression methods (such as PNG or GIF) allow us to get good compression ratio with fast compression and decompression. In this article, we propose semi-adaptive dictionary based compression method, which is a modification of the well-known LZW method. The method allows user to divide the whole image into rectangular blocks and process and transfer coded blocks separately from each other. Despite the small block size, our method gives similar compression ratio not any worse than GIF or PNG without blocking of the image.

***Keywords**: Dictionary-based, semi-adaptive, compression, maps, personal navigation.*

1. INTRODUCTION

Map images are needed in personal navigation and other location-based applications. Typical map image contains high spatial resolution but a limited number of colours as shown in Fig. 1. The images are usually of huge size, and thus, needs to be compressed for efficient storage.

Best compression results have been achieved by *context-based statistical modelling* with *arithmetic coding* without any loss in the image quality [11]. The most recent binary image compression standards JBIG [9] and JBIG-2 [8] are examples of such approach. Compression of about 25:1 can be achieved for a typical good quality noiseless map image [6].

Map image storage system has been proposed in [5] using JBIG as the basic compression method, and by tiling the image into blocks and implementing direct access to the compressed file. The method can be used for dynamic construction of maps from smaller fragments retrieved real-time on-the-demand basis as proposed in [7].

Dictionary-based compression is an alternative approach to the statistical modelling. This approach has been taken in GIF [12] and in PNG [14], for example. These methods can achieve compression of about 10:1 for typical map images [6].

Although JBIG is superior in compression performance, the dictionary-based compression has two advantages: Firstly, the decompression can be about 10-20 faster than that of the context-based modelling and arithmetic coding. Secondly, JBIG can compress only binary images, and thus, the maps must therefore be divided into binary layers before compression. When the image contains lots of colours, the layer separation can be time-consuming and weaken the compression performance. Because of these reasons, the dictionary-based methods are useful in some applications.

We study the dictionary-based compression with application to map images. We consider the case when the image is divided into smaller blocks, which are compressed separately. The number of pixels per block can be relatively small and, therefore, the model has less time to adapt to the input image. This can weaken the compression performance in methods such as GIF and PNG, which uses dynamic modelling.

In this paper, we propose semi-adaptive modelling scheme for the LZW data compression algorithm with application to map images divided into smaller blocks. The proposed compression scheme consists of three stages. In the first stage, the image is analysed and a global dictionary is built for the entire image by using the LZW algorithm. In the second stage, the dictionary is pruned so that it will contain only most frequently used pixel sequences. The dictionary is stored in the compressed file. In the third stage, the input blocks are compressed separately using the global dictionary, which is applied in static manner.

We will show by experiments that the proposed method outperforms the traditional dynamic dictionary construction in the case of small block sizes. The compression performance remains relatively constant as a function of the block size, whereas the dynamic modelling starts to lose its effect for block size smaller than 50×50.

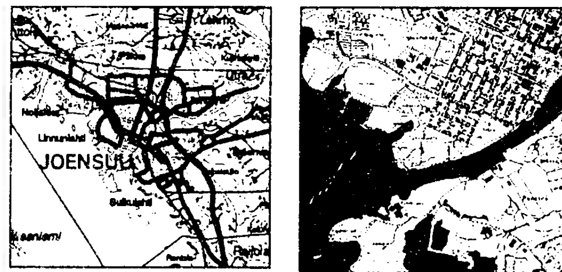

Figure 1. Sample map images: a road map of scale 1:100 000 (left), and a topographic map of scale 1:20 000 (right).

2. PERSONAL NAVIGATION

Personal navigation is a service, which helps people to navigate on work-related and leisure journeys, to choose the

route and mode of transport necessary to reach a particular destination, and to find the service or product that they desire. Usually the guidance supposed to be implemented on the basis of mobile multimedia and it would be available in both out-door and in-door environments. It should be possible to access the services using mobile devices like mobile phones, pocket PCs and traditional PCs via Internet.

2.1. Map service

One of the main fields of the personal navigation service is the map service. Digital maps provide visual view on a given geographic location that can be used for application dealing with spatial data in personal navigation. We can select user-specific views of the map server for different applications. The main goal is to have the maps available in real-time and independent of the location of the user. The map service processes by the next scheme: the user receives his coordinates from the GPS, and sends them to the map image server. After receiving the request the server sends the necessary map.

2.2. Computational resources

Let us consider the case when the user uses the map service in out-door environment. It means that the user has to use a portable device (like a notebook or a pocket PC) with appropriate positioning device and wireless internet-connection. Here arise several problems.

Firstly, it is necessary to say that the storage size of a map is huge. For example, electronic library of Finnish road maps of the resolution 1:250 000 takes the space of entire CD (over 600 Mb) in uncompressed form [4]. At the same time, the capacity of modern communication channels are limited and these limits depend on the type of communication. The GPRS channel has capacity of 56 Kbit per second, while GSM channel has capacity only of 9600 bits per second. In the case when the device (like a pocket PC) operates by an infrared port, then the input channel capacity is limited by 19200 bits per second. On the other hand, the portable viewing device, such as pocket computer, have about 32 Mb of storage space (for everything, including operation system, programs and data), which can be expanded by about 96 Mb through using compact flash memory cards [4]. The size of the map images can therefore be a bottle-neck. That is the main reason to compress map images before the data will be transmitted to the user.

2.3 Block-segmentation of the image

The map service makes its own demands to the map image compression technique. The first place with the compression ratio divides here the quick and low machine resource demandable decompression, because the compression can be processed on powerful servers, but the decompression of the image will be processed on the devices with comparatively low computational resources.

The existing compression techniques have a drawback that the entire image must be decompressed in memory before the image can be presented to the user [4]. This can be a problem, as the device may not have sufficient resources for real-time decompression of the whole image.

One solution to this problem is block segmentation. The image is divided into $b \times b$ non-overlapping rectangular blocks before the compression, and each block is compressed separately as proposed in Figure 2 [4]. The compressed blocks are stored in the same file, and an index table is stored in the header of the file. When the compressed image is accessed, a block index table indicating the location of the block in the compressed file, can be constructed. This provides direct access to the compressed image file, and therefore, enables efficient and independent decompression of particular image fragment.

The block size is a compromise between compression ratio and decoding delay. If very small block size is used the desired part of the image can be reconstructed more accurately and faster. The index table itself requires space and the overhead is relative to the number of blocks.

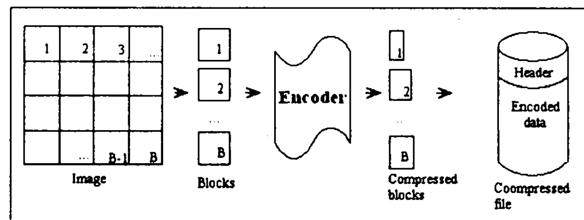

Figure 2. Block-segmentation of the map image.

3. SEMI-ADAPTIVE DICTIONARY BASED COMPRESSION

The compression theory has two main subsections: statistical and dictionary based compression algorithms [3]. Let us closer consider the family of dictionary-based algorithms.

3.1. Dynamic vs. Semi-adaptive

There are three types of dictionary based compression models. They are: static, semi-adaptive and dynamic. The static algorithms are the simplest and the oldest members of this family. They use for encoding a static, predefined dictionary, which is not adapted to the input file. Unfortunately, their behaviour is unacceptable, as we can't create a static dictionary, which will compress all different map images. So let us consider more closely the other two modelling schemes: dynamic and semi-adaptive.

Semi-adaptive dictionary based compression method uses a two-pass algorithm, where the first pass is used for creating dictionary according to input stream, and the second performs the actual compression. The dynamic dictionary based compression uses the one-pass algorithms, which modifies operators and/or attributes according to the input stream. The dictionary in the dynamic case is created online during the compression.

That is the reason why the dynamic algorithm must update the model according to the input stream. On the other hand, semi-adaptive dictionary based compression needs to store the dictionary in the compressed file, whereas the dynamic modelling reconstructs it during the decompression (see Figure 4). The positive and negative sides of both modelling methods are summarized as follows:

Semi-adaptive decompression	Dynamic decompression
–Side information needed	+No side information needed
+No updating of the model during decompression	–Updating of the model during decompression

3.2. Semi-adaptive modification of the LZW algorithm

LZW is a reasonably good compression algorithm, which can adapt itself to the changes in the input stream [15]. During LZW compression the input stream is parsed into phrases, where each phrase is the longest matching phrase seen previously. Each phrase (without its last character) is encoded as an index to the dictionary. The main achievement of this approach is that it is from the family of adaptive coders that let it not to store the dictionary into the file, but create it during decompression process. It is very important, because the dictionary can easily reach very huge size. On the other hand, the compression ability of the LZW algorithm decreases if we use it for compression maps with small number of colours (binary images for example), or texts with small number of different used symbols.

We present a modification to the LZW, which is denoted here as *LZWsem* (*LZW semi-adaptive*). The algorithm has two phases: in the first phase it builds a dictionary, and the actual output of the indexes is started in the second phase. The dictionary is built by using the same algorithm as in the LZW. During the second stage, the LZWsem uses also the same parsing technique as LZW but it can use all entries in the dictionary, also those ones that appear later in the image. This also means that the dictionary must also be coded.

3.3. Implementation of the algorithm

The implementation of the semi-adaptive method can be divided into several parts as follows:

1. Create an initial dictionary by the LZW algorithm,
2. Prune the dictionary,
3. Store the final dictionary in the compressed file,
4. Encode the input file by using the final dictionary.

The first part is creation of *initial dictionary*. It means that at the first time, the algorithm processes the adaptive LZW coding process, and create dictionary so called as initial dictionary. The indexes, however, are not yet coded. At the next step, the algorithm prunes the initial dictionary. The third step is in storing all necessary side information, as the file header, dictionary and so on into the compressed file. At the final step, the algorithm codes the input image by using the final dictionary.

3.4. Creation of the initial dictionary

For creating the initial dictionary we use the original LZW algorithm. On this stage we predefine the number of LZW tree nodes per each block. The direction of the compression is processed as it is shown in the Figure 3. The number of nodes per each block is the maximum number of nodes, divided by the number of blocks. The pseudocode of this process is shown in the Figure 4. The *MaxNodeIndex* is the upper limit of the LZW tree node indexes, and *MaxBlockNodeNumber* is the number of nodes per block.

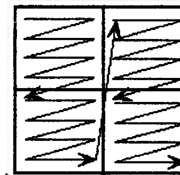

Figure 3. The direction of the image processing in the second variant

```
MoveFilePointerToBlockStart
Count←0
MaxBlockNodeNumber = MaxNodeIndex/NumberOfBlocks
String←ReadSymbol(InputImage)
While(Not end of image block)
{
Symbol←ReadSymbol(InputImage)
  if(String+Symbol∈InitialDictionary){
      String←String+Symbol
  }
  else{
      AddToDictionary(String+Symbol)
      Count++
      if(Count>=MaxBlockNodeNumber) then break
  }
  String←Symbol
}
```

Figure 4. The pseudocode of the block-to-block dictionary creating.

3.5. Pruning of the initial dictionary

Transformation of the initial dictionary into the final dictionary consists of deleting the most of elements from the initial dictionary. Let us define a *cover* of the input image as the sequence of the dictionary elements, which cover the image without intersections. So the process consists of two stages:

1. Cover creating.
2. Pruning of the cover.

The algorithm of the first phase is described in Figure 5. All elements from the initial dictionary are divided into two groups: symbols sequences included in the cover, and symbols sequences not in the cover.

```
Read( FirstSymbol )
CurrentString ← FirstSymbol
CurrentStringIndex ← 1
While( Not end of the input stream)
{
Read(Symbol)
if(CurrentString + Symbol ∉ InitialDictionary)
      {
        Index(CurrentString) ← CurrentStringIndex ++;
        CurrentString ← Symbol
      }

else
      {
      CurrentString ← CurrentString + Symbol
      }
}
```

Figure 5. Pseudo code for the creation of the cover.

At the second stage of this phase we start to prune the initial dictionary. For this operation we need a criterion for each element from the initial dictionary, whether to keep it or to get rid from it. The main principle of this criterion was that each element of the final dictionary will be represented in the header of the compressed file and therefore will require additional space. Among several variants of the criterion the simplest one was chosen. During the first stage we collect the cover statistics of each element from the initial dictionary. The criterion is: if an element appears in the cover of the input image more than once it is kept in the dictionary. Otherwise it is deleted.

3.6. The final dictionary structure

We consider two alternative methods to store the final dictionary elements:

1. To keep the tree structure of the LZW tree; the final dictionary is stored as pointer to the parent node + symbol.
2. To get rid of the tree structure and store the final dictionary as a simple codebook.

In the first case, the element of the dictionary is consisting from: "parent + symbol". Size of the parent is calculated from the predefined number of LZW tree nodes. For example, if LZW tree consists of 512 nodes, then it is necessary to use 9 bits to encode the index of the nodes ($\log_2 512=9$). The code bit length of the symbol depends from the number of colours. So if the image has 8 colours, it is necessary to use at least 3 bits to encode the index of colour ($\log_2 8=3$). The final formula for the size of the transformed information is:

$(\log_2 ColourNumber + \log_2 NodesNumber)\cdot NodesNumber$ bits. In the second variant the formula for the size of transmitted information is:

$\sum_{i=1}^{UseNodesdNumber} DepthOfNode_i \cdot \log_2 ColoursNumber$ bits, where

UsedNodesNumber is the number of nodes from the initial dictionary that were used in the cover of the input image, and *DepthOfNode* is the depth of the node in the LZW tree (or length of the phrase, which it represents).

Both variants were tested on several map images, and results of the tests are placed in the Table 1. The advantage of the second variant is clear when the number of LZW tree nodes is higher. There is a tendency that the difference between size of transmitted information increases in favour of the second variant.

Table 1. Comparison of both variants for storing the dictionary.

Image	Nodes Number	Colours Number	Used Nodes Number	Variant1 Size	Variant2 Size
vl1.pgm	512	4	201	5632	3198
Vantaa.pgm	512	4	426	5632	5570
Suomi2-q.pgm	512	8	353	6144	5928
hills.pbm	512	2	325	5120	3505

3.7. Storing the final dictionary

The first information, which is necessary to read the final dictionary, is the number of sequences and the lengths of each sequence. As the number of sequences is about several hundreds, they are encoded by using Huffman coding. The process of storing the final dictionary in the compressed file can be divided into several stages:

1. Output the length of the dictionary (the number of dictionary elements after two stages of pruning).
2. Calculate statistics for all lengths of the final dictionary sequences.
3. Create a Huffman tree for the lengths according to the statistics.
4. Store the Huffman tree structure in the compressed file.
5. Output elements of the sequences.

At the first stage, encoder outputs the number of elements in the dictionary.

The second stage consists of two sub-stages. At the first sub-stage it is necessary to find the maximum length from all sentences from the final dictionary. The second is to calculate how often each length appeared in the final dictionary.

At the third and fourth stages, the encoder codes lengths of the dictionary elements by using Huffman coding. It creates Huffman tree and outputs all information that is necessary for reconstruction of the Huffman tree during decompression.

The last stage outputs the actual dictionary. The dictionary is represented as a sequence of characters and encoded by the LZSS algorithm [15]. The general scheme of the header of the compressed file is placed in the Figure 6.

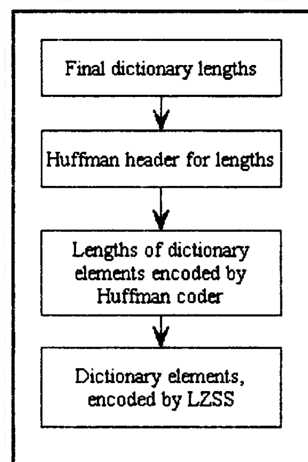

Figure 6. General scheme of storing the dictionary.

3.8. Compression step

When all necessary information of the dictionary is outputted, it is time to compress the input image. This algorithm is a two-stage process:

1. Calculate the final dictionary elements statistics.
2. Output the compressed information accordingly to the calculated statistics.

The idea of the first stage is to find the modified cover of the input image according to the final dictionary elements. During the first stage, the first version of the cover is cre-

ated. But during the second stage some of the elements from the cover are deleted. In the compressed file they will be replaced by a new sequence of codes. The final dictionary does not have tree structure, but some of the relations are kept. The structure of the final dictionary can be described in the following manner: each element has a *parent* and so-called *tail*: a sequence of symbols, which belonged to all initial dictionary elements that were between the parent and the current node (see Figure 7 and Figure 8 consequently). The pseudocode of the current stage is placed in Figure 9.

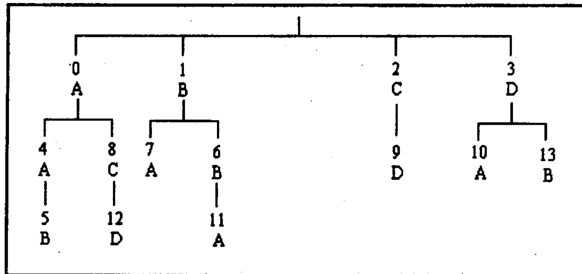

Figure 7. LZW tree for input string "*AAABBACDBBAACDA*".

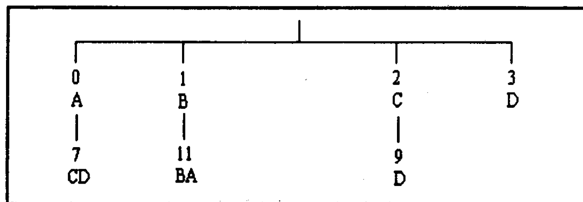

Figure 8. LZW tree for input string "*AAABBACDBBAACDA*" after first stage of pruning.

```
For(phase = 1; phase <= 2; phase++)
{
 String←ReadSymbol(InputImage)
 While(Not end of input stream)
 {
 Symbol←ReadSymbol(InputImage)
  if(String+Symbol∈InitialDictionary){
      String←String+Symbol
  }
  else{
if(String∈FinalDictionary){
if(phase=1) then GetStatistics(CodeOf(String))
      else Output(CodeOf(String))
      }
      else{
         Output(ParentOf(String))
         Output(TailOf(String))
      }
  }
  String←Symbol
 }
}
```

Figure 9. Pseudocode of the encoding in semi-adaptive algorithm.

This pseudocode is the same in both phases, only the function "*Output*" in the first case means calculating statistics of the dictionary elements, and in the second case it means outputting the encoded information into the compressed file. After the first phase, we create the Huffman tree for the final dictionary. The information about the Huffman tree is placed into the compressed file. Then we encode the input image. Each index from the final dictionary is replaced by a corresponding sequence of bits.

3.9. Decompression

The decompression process consists of three steps:

1. Decode the dictionary
2. Looking for the necessary block
3. Decoding of the found

At the first stage, the decoder reads the number of dictionary elements, their lengths, allocates memory for them, and then reads the symbols of each dictionary element. At the next stage, the decoder calculates the number of bytes that is necessary to offset from the beginning of the file to the beginning of the image block. When all necessary procedures are done the decoder starts the decompression using standard LZW decompression routine expect that the dictionary is static and, therefore, not updated during the process.

4. EXPERIMENTS

We compress a set of topographic map images originating from the NLS topographic database [13]. The first image set consists of binary images and the second set of greyscale images. The set of experiments was provided to receive information about the behaviour of the previously described semi-adaptive (LZWsem) algorithm and the adaptive LZW algorithm in the case of block segmentation. As the standard GIF format does not have such option as image tiling, a modified version with this option was implemented. For the experiments we had taken five binary images with size 1000×1000 pixels, and four greyscale images with size 1024×1024 pixels. During the experiments the block segmentation for each image was performed.

The binary images were processed with the block segmentation of sizes: 50×50, 100×100, 200×200, 250×250, 500×500 and 1000×1000 (the whole image compression). For comparison were taken from one side such widely known dictionary based algorithms as PNG, GIF, TIFF and JBIG (here was used the MISS [5] coder, which was processed for binary images).

The greyscale images were processed the block segmentation sizes: 64×64, 128×128, 256×256, 512×512 and 1024×1024 pixels in one block. The results of experiments were summarised and were taken average from all files of each type. These results are shown in the Table 2. The average compression rate for each test image is shown in the Table 3. The dependencies between the compression ratio and the image size are shown in the Figure 10 and 13 (for binary images and for greyscale images consequently).

Table 2. Average sizes of the compressed files from test series.

Binary images						
	S/A	Adaptive	GIF	PNG	TIFF G4	JBIG
50x50	29300	46327	X	X	X	14143
100x100	29491	38196	X	X	X	11965
200x200	29579	33747	X	X	X	10857
250x250	29245	32973	X	X	X	10305
500x500	29272	31017	X	X	X	10616
1000x1000	28882	29993	31417	29537	17934	10215

Greyscale images					
	S/A	Adaptive	GIF	PNG	TIFF Deflate
64x64	302130	312253	X	X	X
128x128	296874	299802	X	X	X
256x256	287727	291508	X	X	X
512x512	286159	289216	X	X	X
1024x1024	283625	288423	317034	318645	315238

Table 3. Average compression ratios.

	S/A	GIF	PNG	TIFF	JBIG
Image 1	3.313	3.18	3.46	5.12	8.55
Image 2	6.07	6.15	6.45	11.05	8.55
Image 3	3.97	3.41	3.61	6.53	12.30
Image 4	8.86	9.15	9.9	26.99	38.18
Image 5	2.64	2.54	2.77	4.22	6.84
Image 6	7.94	7.54	6.86	7.61	X
Image 7	3.30	3.14	3.11	3.16	X
Image 8	2.60	2.42	2.47	2.43	X
Image 9	3.11	2.90	2.91	2.91	X
Average	4.64	4.49	4.62	7.78	14.88

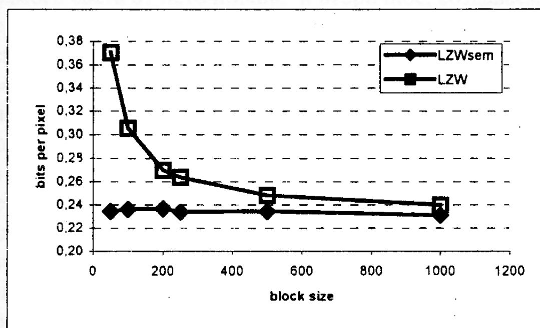

Figure 10. Dependency between bit rate and block size for binary images.

5. CONCLUSION

The article was dedicated to the problems of modern map image compression and possible ways to solve them. In the article we proposed semi-adaptive modification of the LZW algorithm for compressing map images that were divided into smaller blocks. The proposed method as well as GIF, PNG, TIFF and JBIG (for binary images only) were applied to the set of test images.

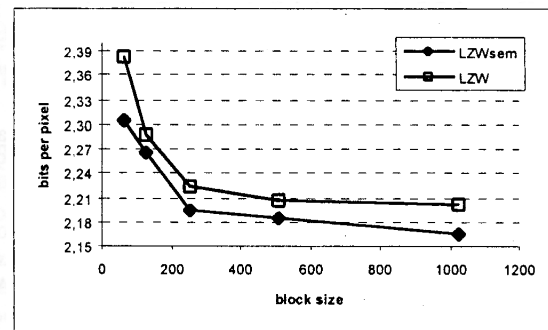

Figure 11. Dependency between bit rate and block size for greyscale images.

The results show us that the decrease of the block size will result in a rapid increase of the resulting code size in the case of the adaptive methods such as the LZW algorithm. These results also show us that in the case of the semi-adaptive method, the decrease of the block size has less influence on the resulting code size. So from the experiments it follows that an increase in the number of blocks by four times leads to an increase the compressed file size by 2-9 %, on average.

For the set of binary images, JBIG has the best compression results, TIFF G4 took the second place, and the semi-adaptive LZW method has the same compression performance as GIF and PNG compressors. For the set of greyscale images, it showed us the comparable performance with GIF, PNG and TIFF compressors.

6. REFERENCES

1. Ageenko E and Fränti P, "Compression of large binary images in digital spatial libraries", *Computers & Graphics* 24 (1): 91-98, February 2000.
2. Akimov A, "Dictionary-based compression of map images", *M.Sc. thesis, Dept. of Computer Science, Univ. of Joensuu,* Finland: December 2001.
3. Bell T, Cleary J, Witten IH, "*Text compression*", Prenitce-Hall, New Jersey, 1990.
4. Fränti P, "*Image Compression*", Lecture Notes, Department of Computer Science, University of Joensuu, 2000, (http://cs.joensuu.fi/pages/franti/comp/comp.doc)
5. Fränti P, Ageenko E, Kopylov P, Gröhn S and Berger F, "Map image compression for real-time applications", *Spatial Data Handling 2002 Symposium (SDH'02)*, Ottawa, Canada, July 2002.
6. Fränti P, Kopylov P and Ageenko E, "Evaluation of compression methods for digital map images", *IASTED Int. Conf. on Automation, Control and Information Technology (ACIT 2002)*, Novosibirsk, Russia, pp. 401-405, June 2002.
7. Fränti P, Kopylov P and Veis V, "Dynamic use of map images in mobile environment", *IEEE Int. Conf. on Image Processing (ICIP'02)*, Rochester, New York, USA, September 2002. (to appear)

8. Howard PG, Kossentini F, Martins B, Forchammer S and Rucklidge WJ, "The emerging JBIG2 standard," *IEEE Trans. Circuits and Systems for Video Technology*, 8: 838-848, 1998.

9. ISO/IEC, Final Committee Draft for ISO/IEC International Standard 14492, 1999. (http://www.jpeg.org/public/jbigpt2.htm)

10 Kopylov P and Fränti P, "Context tree compression of multi-component map images", *IEEE Data Compression Conference* (*DCC'02*), Snowbird, Utah, USA, pp. 212-221, April 2002.

11. Langdon GG, Rissanen J, "Compression of black-white images with arithmetic coding", *IEEE Trans. Communications* 29: 858-867, 1981.

12. Miano J, "*Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*", ACM Press, Addison-Wesley, Boston, 1999.

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

14. Roelofs G, "*PNG: The Definitive Guide*", O'Reilly & Associates, Cambirdge, MA: 1999.

15. Salomon D, "*Data compression: complete reference*", Maple-Vail Manufacturing Group, York, 2000.

16 Storer JA and Szymansky TG, "Data compression via textual substitution", *Journal of the ACM* 29:928-951, 1982.