Chapter 6:

Conclusion

6. Conclusion

As a conclusion to this dissertation report, an overview of what have been accomplished is presented. A summary of the work accomplished and results are discussed in Section 6.1. Suggestions for future work are presented in Section 6.2.

6.1 Summary of work accomplished and results

Lossless compression of road map images was the focus of this dissertation. The use of a statistical technique called arithmetic coding with adaptive context-based statistics modeling to compress the road map images was investigated.

Arithmetic coding can be divided into two parts, the model and the coder. For a given model, arithmetic coding will give close to optimum compression. As pointed out by Moffat, the coder portion serves as an "engine" for the compression system, while the modeling is the "intelligence" which drives the system (Moffat *et al.*, 1998). Improvements to the model will yield improvement to the compression effectiveness, in terms of reduced size of the compressed data. Improvements to the coder are mainly concerned with the compression efficiency, that is, a reduction in time or memory usage. As such, the implementation focused on the modeling portion. For the coder, the arithmetic coder implementation by Moffat was incorporated as the engine of the compression program.

Two different approaches to adaptive context modeling were used, bitplane coding and PPM. Implementation was successfully done using the C programming language. Testing was performed using a set of fifteen road map images of various sizes and colours. Table 6.1 provides a summary of the results in terms of the total file sizes.

Table 6.1: Comparison of total file sizes (in bytes)

original raw data	GIF (image data only)	bitplane coding	PPM max. order = 4	PPM variable max. order	combined method
3,080,740	305,052	172,124	113,449	121,860	123,474

In bitplane coding, the image was decomposed into individual bitplanes, which were then compressed separately as binary images using a binary arithmetic coder and fixed context modeling. Bitplane reduction was applied to minimize the number of bitplanes that need to be compressed. Several types of neighbourhood templates were used to find the context. It was found that the order-12, 2-norm template gave the best overall compression results, as far as the set of test images were concerned. Compared to the original raw data, a total file size reduction of 94.4% was achieved. Compared to GIF, a commonly-used file format, the total file size reduction was 43.6 % A possible enhancement using reflected Gray codes was also tried, but this failed to yield a better compression overall.

The second method investigated was PPM, where variable order context modeling was used together with an arithmetic coder. In PPM, the compressor initially attempts to encode a symbol using the predefined maximum order. If is fails to do so, an escape symbol is encoded and a lower order is tried. The rationale for this is that during the early stages of compression, the higher order statistics have yet to become reliable, and thus the lower order statistics are used (Cleary & Witten, 1984). One issue with PPM is the probability to assign to the escape symbol. Several types of escape method were investigated, namely method A, B, C and D. It was found that method A gave the best compression overall for the set of images, followed by method D, C and B. Among the two types of neighbourhood templates that were used, the 2-norm template gave the better result.

Several other enhancements for PPM were also tried. The effectiveness of update exclusion was confirmed based on the test results. For the frequency count scaling value, it was found that reducing the value improved the compression only marginally. A small count scaling value (4,096) when compared to a very large value (524,288) yielded only a total size difference of 0.5%. In fact, if the value is made smaller, at some point the compression effectiveness will start to suffer.

As far as the maximum order is concerned, it was found the larger the value, the more effective the compression. Using a maximum order of 4, and the best settings obtained, PPM was able to achieve a total file size reduction of 96.3% compared to the original raw data. Compared to GIF, the reduction achieved was 62.8%. PPM therefore compressed the files more effectively than bitplane coding. Both, however, proved more effective than GIF.

The problem with this implementation of PPM is the large amount of memory space required to store the probability tables. This is because arrays were used for the tables. At the maximum order of 4, the memory requirement becomes enormous when the number of colours in an image increases. For example, when the number of colours is 32, about 213 Mbytes are required, compared to abut 7.1 Mbytes when the number of colours is 16. If the memory space required is more than the system's physical memory, then hard disk trashing occurs, which drastically slows down the program.

Two attempts were made to constrain the memory requirement when using PPM. The first is to simply use a smaller value for the maximum order when the number of colours increases. For example, when the number of colours is up to 16, the maximum order of 4 is used, and when it is between 16 to 32, the maximum order of 3 is used instead. This

constrains the memory requirement to about 7 Mbytes. The resulting overall compression was only slightly worse than PPM with a fixed maximum order of 4, and still significantly better than bitplane coding, as shown in Table 6.1.

The second attempt to constrain the memory requirement of PPM is to use a combined method, where bitplane coding and PPM is used to compress different planes separately. This approach taken is unique and never tried before. As far as bitplane coding is concerned, the memory requirement for its probability tables is relatively small. Therefore, the following scheme was tried. The best settings found for both methods were used. If the number of colours is more than 16, bitplane coding is used to compress the additional bitplanes. If it is less than 16, only PPM is used. This also constrains the memory requirement to about 7 Mbytes. The resulting overall compression is, unfortunately, slightly worse than the variable maximum order PPM. The set of road map images contain images of up to 32 colours only. It would be interesting to compare the results with the variable maximum order PPM when used for images with a larger number of colours (33 to 256).

For execution speed, only a rough comparison was made since the program implementations were not optimized for speed. It was found that bitplane coding was at least twice faster than PPM and the combined method.

As a summary, the implemented methods were able to compress the road map images, and they did so more effectively compared to GIF.

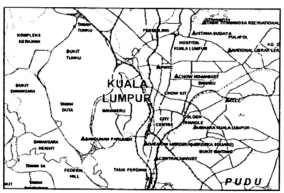
6.2 Future work

Several suggestions of possible future work and improvements are listed below:

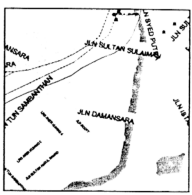
1. The use of arrays to store the probability tables in PPM incurs a large requirement for memory space. Previous work in text compression using PPM employed data structures such as hash tables, lists or trees (Cleary & Witten, 1984; Moffat, 1990; Howard & Vitter, 1992). The possibility of using such data structures can be investigated for the case of image compression. However, when using such data structures, the execution speed is expected to suffer, so the tradeoff can also be investigated.
2. The use of the combined approach can be investigated further for the case of map images containing a larger number of colours (33-256) compared to the existing set of images. Although the compression suffers a little, this approach sounds attractive because the memory requirement is constrained without requiring the use of special data structures mentioned in (1).
3. More sophisticated escape methods for PPM, such as PPMX (Witten & Bell, 1991) and PPM* (Cleary *et al.*, 1995), can be tried or other escape mechanisms can be devised for road maps images.
4. In this implementation, the escape symbol was not included in the probability tables. Instead, its contribution to the symbol probability was calculated explicitly when the function call to the arithmetic coder was made. In hindsight, perhaps it is better to create an entry for it in the probability tables, although the case for this is not quite clear.

Appendix A: Road map images

Note: All the images were obtained from the websites indicated in Table 3.10 on 3rd February, 2003.



a400600

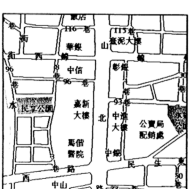


b541541

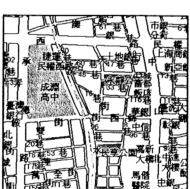
c541541

d541541

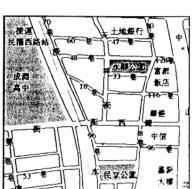
e541541



f400400

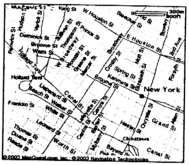


g400400

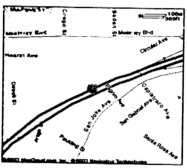


h400400

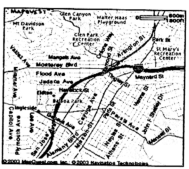
i309356



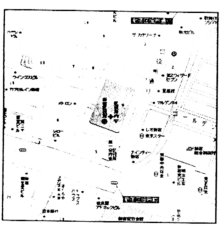
j309356



k309356

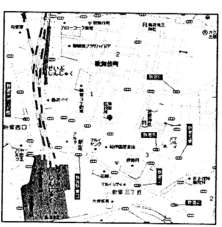


l309356



m500500

n500500



o500500

Appendix B: RAW and JASC-PAL file formats

RAW File Format

The RAW file format (Laterre, 1998; *Graphics Formats Specifications*, undated) is a flexible format that can be used to transfer files between different applications and platforms. Unlike other formats, a RAW file has no header information (such as width, height, etc.) to describe the data. The details must be supplied by the user in order to read the data. The content of a RAW file is just the pixel data, in the case of images. It consists of a stream of bytes describing the colour information in the file. Each pixel is represented by a binary-coded number. Paint Shop Pro allows a file to be saved in either 8 or 24 bits per pixel, depending on the number of colours in the source file.

JASC-PAL Palette File Format

A palette file is used to store the colour table of a palette image. Paint Shop Pro uses its own JASC-PAL format (Jong, undated; *JASC Palette File Format*, undated). The file is an ASCII text file. The description of each line in the file is shown in Table B.1.

Table B.1: JASC-PAL palette file format

Line Number	Description
1	Contains the string 'JASC-PAL'
2	Contains the string '0100'
3	Indicates the number of palette entries in the file
4 onwards	Each line corresponds to a palette entry. It starts with the value for the red component, then green, then blue. The values are separated by single spaces, so each line has a total of 2 spaces. The palette entries continue until the end of the file.

Appendix C: Calculation of GIF file's image data size

Figure C.1 shows the structure of the GIF file (CompuServe Inc., 1990).

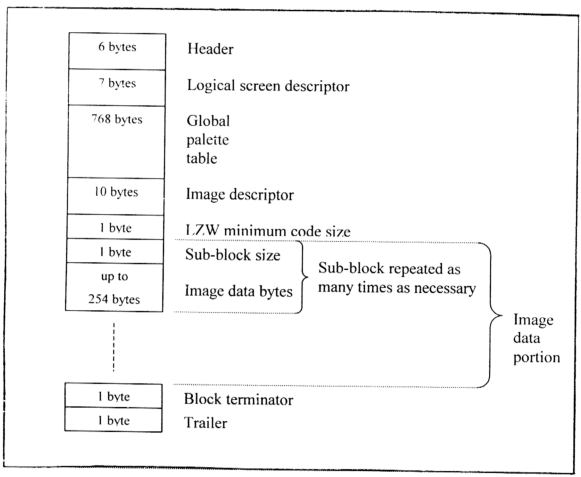


Figure C.1: Structure of a GIF file

The compressed image data is divided into sub-blocks, where each sub-block is up to 255 bytes, including a byte to indicate the sub-block's size, i.e. each sub-block has 254 data bytes, except the last sub-block, which may be less than 254 data bytes.

The size of the image data portion of the GIF file is calculated as following. Let f denote the total file size. The size of the non-image data portion is altogether 794 bytes.

The number of sub-blocks, b, is given by

$$b = \lceil (t-n)/255 \rceil$$

where $\lceil x \rceil$ denotes the smallest integer equal or greater than x. Therefore, the size of the image data is given by f – 794 – b. Table C.1 shows the list of test images GIF file's size and the calculated image data size.

For example, if the GIF file is 20,085 bytes, b = 76. The image data size is thus 20,085 - 794 -76 = 19,215 bytes.

Table C.1: Test image GIF file size

file name	file size [bytes]	image data size [bytes]
a400600	20,085	19,215
b541541	17,767	16,906
c541541	25,727	24,835
d541541	20,483	19,611
e541541	31,353	30,439
f400400	13,743	12,898
g400400	21,597	20,721
h400400	14,967	14,117
i309356	11,169	10,334
j309356	17,908	17,046
k309356	9,730	8,900
l309356	19,763	18,894
m500500	20,734	19,861
n500500	35,656	34,725
o500500	37,488	36,550

Appendix D: Memory requirement in PPM

The memory requirement for the probability tables in this implementation of PPM is given by the Table D.1 below (in bytes).

Table D.1: Memory requirement for probability tables in PPM

no. of symbols	maximum order				
	0	1	2	3	4
3	23	92	299	920	2783
4	29	145	609	2465	9889
5	35	210	1085	5460	27335
6	41	287	1763	10619	63755
7	47	376	2679	18800	131647
8	53	477	3869	31005	248093
9	59	590	5369	48380	435479
10	65	715	7215	72215	722215
11	71	852	9443	103944	1143455
12	77	1001	12089	145145	1741817
13	83	1162	15189	197540	2568103
14	89	1335	18779	262995	3682019
15	95	1520	22895	343520	5152895
16	101	1717	27573	441269	7060405
17	107	1926	32849	558540	9495287
18	113	2147	38759	697775	12560063
19	119	2380	45339	861560	16369759
20	125	2625	52625	1052625	21052625
21	131	2882	60653	1273844	26750855
22	137	3151	69459	1528235	33621307
23	143	3432	79079	1818960	41836223
24	149	3725	89549	2149325	51583949
25	155	4030	100905	2522780	63069655
26	161	4347	113183	2942919	76516055
27	167	4676	126419	3413480	92164127
28	173	5017	140649	3938345	110273833
29	179	5370	155909	4521540	131124839
30	185	5735	172235	5167235	155017235
31	191	6112	189663	5879744	182272255
32	197	6501	208229	6663525	213232997

The computation is done as follows. Let

s = number of symbols (colours)

m = the maximum order

The C compiler allocates 1 byte for the variable of type *char* and 2 bytes for the variable of type *short int*. So, the size of each probability table is 6s +5 bytes, as shown below:

array	type	array size	bytes
freq[ ]	*short int*	s+1	2s+2
cum_freq[ ]	*short int*	s+1	2s+2
char_to_index[ ]	*char*	s	s
index_to_char[ ]	*char*	s+1	s+1
		total:	6s+5

The number of possible contexts for each order is given by s^c, where c is the context order. Therefore, the memory required for the probability tables is given, in bytes, by:

$$\sum_{i=0}^{m} (6s+5)\, s^i$$