哈尔滨工业大学

算数编码实验报告

姓名：金富源

学号：1170300404

学院：计算机科学与技术学院

教师：刘岩

算术编码作业内容：

1、 给出算术编码实现的详细原理。
2、 编制编解码程序
3、 设计并实现自适应算术编码（选做）

1 实验原理

早在 1948 年，香农就提出将信源符号依其出现的概率降序排序，用符号序列累计概率的二进值作为对芯源的编码，并从理论上论证了它的优越性。1960 年，Peter Elias 发现无需排序，只要编、解码端使用相同的符号顺序即可，提出了算术编码的概念。Elias 没有公布他的发现，因为他知道算术编码在数学上虽然成立，但不可能在实际中实现。

算术编码的基本原理是将编码的消息表示成实数 0 和 1 之间的一个间隔（Interval），消息越长，编码表示它的间隔就越小，表示这一间隔所需的二进制位就越多。

算术编码用到两个基本的参数：符号的概率和它的编码间隔。信源符号的概率决定压缩编码的效率，也决定编码过程中信源符号的间隔，而这些间隔包含在 0 到 1 之间。编码过程中的间隔决定了符号压缩后的输出。

算数编码是一种非常有用的无损信源压缩编码方式，也是一种熵编码的方式。
一般的熵编码方式，通常是将信源消息分割成一个符号序列，然后对单个的符号进行编码。而算数编码则将整个待编码的符号序列映射到一个位于[0,1)实数区间。
算术编码的基本原理是将编码的消息表示成实数 0 和 1 之间的一个间隔（ Interval），消息越长，编码表示它的间隔就越小，表示这一间隔所需的二进制位就越多。利用这种方法，算数编码可以将压缩率无限的接近于数据的熵值，从而获得理论上的最高压缩率。
在静态的算数编码中，需要提前统计信源符号的概率分布。在编码时，从实数区间 [0,1)开始，按照符号的频度将当前的区间分割成多个子区间。根据当前输入的符号选择对应的子区间，然后从选择的子区间 中继续进行下一轮的分割。不断的进行这个过程，直到所有符号编码完毕。对于最后选择的一个子区间，输出属于该区间的一个小数。这个小数就是所有数据的编码。

在给定符号序列的算数编码过程如下：
（1） 编码器开始时，将“当前间隔” [L,H)设置为[0,1)
（2） 对每个输入符号，编码器按照下面的步骤（3）（4）进行编码操作
（3） 编码器将当前间隔分为“子间隔”，每个符号一个“子间隔”，“子间隔”的相对大小同符号的概率成正比
（4） 编码器选择与当前输入符号相匹配的“子间隔”，并将其作为新的“当前间隔”

解码过程如下：

（1） 将编码得到的小数序列作为“当前编码”

（2） 寻找“当前编码”所在[0,1)区间的位置，即找到位于哪一个符号的“子区间”上，就将该符号作为当前译码得到的符号

（3） 计算“当前编码”位于该符号“子区间”中的对应比例的位置，得到新的“当前编码”。计算方法为：当前编码 – low（当前子区间）

$$\frac{当前编码 - low_{当前子区间}}{high_{当前子区间} - low_{当前子区间}} = 新的当前编码$$

（4）重复（2）（3）步骤，直到译码到终止符号或者到指定字符数目时停止

实验内容:

其主要思想是：

1. 设定初始区间为 [0,1)
2. 根据字符频率来对区间进行非等长划分，字符频率越高，其分得的区间长度越长
3. 将区间进行裁剪，裁剪到待编码字符的区间范围
4. 重复步骤 2 直到所有字符完成编码

实现代码（cpp）：

```
#include "zjw_arithmetic_coder.h"

unsigned long AdaptiveArithmeticCoder::encodeCharVectorToStream(const std::vector<char>& inputByteVector_arg, std::ostream & outputByteStream_arg)
{
    DWord freq[257];
    uint8_t ch;
    unsigned int i, j, f;
    char out;

    // define limits
    const DWord top = static_cast<DWord> (1) << 24;
    const DWord bottom = static_cast<DWord> (1) << 16;
    const DWord maxRange = static_cast<DWord> (1) << 16;

    DWord low, range;
```

```
unsigned int readPos;
unsigned int input_size;

input_size = static_cast<unsigned> (inputByteVector_arg.size());

// init output vector
outputCharVector_.clear();
outputCharVector_.reserve(sizeof(char) * input_size);

readPos = 0;

low = 0;
range = static_cast<DWord> (-1);

// initialize cumulative frequency table
for (i = 0; i < 257; i++)
	freq[i] = i;

// scan input
while (readPos < input_size)
{

	// read byte
	ch = inputByteVector_arg[readPos++];

	// map range
	low += freq[ch] * (range /= freq[256]);
	range *= freq[ch + 1] - freq[ch];

	// check range limits
	while ((low ^ (low + range)) < top || ((range < bottom) && ((range = -int(low) & (bottom
- 1)), 1)))
	{
		out = static_cast<char> (low >> 24);
		range <<= 8;
		low <<= 8;
		outputCharVector_.push_back(out);
	}

	// update frequency table
	for (j = ch + 1; j < 257; j++)
		freq[j]++;

	// detect overflow
```

```
            if (freq[256] >= maxRange)
            {
                // rescale
                for (f = 1; f <= 256; f++)
                {
                    freq[f] /= 2;
                    if (freq[f] <= freq[f - 1])
                        freq[f] = freq[f - 1] + 1;
                }
            }

        }

        // flush remaining data
        for (i = 0; i < 4; i++)
        {
            out = static_cast<char> (low >> 24);
            outputCharVector_.push_back(out);
            low <<= 8;
        }

        // write to stream
        outputByteStream_arg.write(&outputCharVector_[0], outputCharVector_.size());

        return (static_cast<unsigned long> (outputCharVector_.size()));
    }

    unsigned long AdaptiveArithmeticCoder::decodeStreamToCharVector(std::istream &
    inputByteStream_arg, std::vector<char>& outputByteVector_arg)
    {
        uint8_t ch;
        DWord freq[257];
        unsigned int i, j, f;

        // define limits
        const DWord top = static_cast<DWord> (1) << 24;
        const DWord bottom = static_cast<DWord> (1) << 16;
        const DWord maxRange = static_cast<DWord> (1) << 16;

        DWord low, range;
        DWord code;

        unsigned int outputBufPos;
        unsigned int output_size = static_cast<unsigned> (outputByteVector_arg.size());
```

```
unsigned long streamByteCount;

streamByteCount = 0;

outputBufPos = 0;

code = 0;
low = 0;
range = static_cast<DWord> (-1);

// init decoding
for (i = 0; i < 4; i++)
{
	inputByteStream_arg.read(reinterpret_cast<char*> (&ch), sizeof(char));
	streamByteCount += sizeof(char);
	code = (code << 8) | ch;
}

// init cumulative frequency table
for (i = 0; i <= 256; i++)
	freq[i] = i;

// decoding loop
for (i = 0; i < output_size; i++)
{
	uint8_t symbol = 0;
	uint8_t sSize = 256 / 2;

	// map code to range
	DWord count = (code - low) / (range /= freq[256]);

	// find corresponding symbol
	while (sSize > 0)
	{
		if (freq[symbol + sSize] <= count)
		{
			symbol = static_cast<uint8_t> (symbol + sSize);
		}
		sSize /= 2;
	}

	// output symbol
	outputByteVector_arg[outputBufPos++] = symbol;
```

```
            // update range limits
            low += freq[symbol] * range;
            range *= freq[symbol + 1] - freq[symbol];

            // decode range limits
            while ((low ^ (low + range)) < top || ((range < bottom) && ((range = -int(low) & (bottom - 1)), 1)))
            {
                inputByteStream_arg.read(reinterpret_cast<char*> (&ch), sizeof(char));
                streamByteCount += sizeof(char);
                code = code << 8 | ch;
                range <<= 8;
                low <<= 8;
            }

            // update cumulative frequency table
            for (j = symbol + 1; j < 257; j++)
                freq[j]++;

            // detect overflow
            if (freq[256] >= maxRange)
            {
                // rescale
                for (f = 1; f <= 256; f++)
                {
                    freq[f] /= 2;
                    if (freq[f] <= freq[f - 1])
                        freq[f] = freq[f - 1] + 1;
                }
            }
        }

        return (streamByteCount);
}
```
算法：

结和算术编码的要求，该算法只需关注小数部分即可，因为不论是 Fui 还是 Pui 都不会大于等于 1（对于 Pui 0 和 Fui 0 我是单独计算的）；定义加法、减法和乘法（除法因为能力问题没有设计出来，选择使用别的方法代替）

除法的应用场景是在计算序列长度时，所以我选择使用比较的方法。将得到的 Pui 与 0.5 的 n 次方循环比较（效率低下），得到第一个比 Pui 小的 n 就是所求值。

头文件（.h）：

```
#include "zjw_arithmetic_coder.h"

unsigned long AdaptiveArithmeticCoder::encodeCharVectorToStream(const std::vector<char>&
inputByteVector_arg, std::ostream & outputByteStream_arg)
{
    DWord freq[257];
    uint8_t ch;
    unsigned int i, j, f;
    char out;

    // define limits
    const DWord top = static_cast<DWord> (1) << 24;
    const DWord bottom = static_cast<DWord> (1) << 16;
    const DWord maxRange = static_cast<DWord> (1) << 16;

    DWord low, range;

    unsigned int readPos;
    unsigned int input_size;

    input_size = static_cast<unsigned> (inputByteVector_arg.size());

    // init output vector
    outputCharVector_.clear();
    outputCharVector_.reserve(sizeof(char) * input_size);

    readPos = 0;

    low = 0;
    range = static_cast<DWord> (-1);

    // initialize cumulative frequency table
    for (i = 0; i < 257; i++)
        freq[i] = i;

    // scan input
    while (readPos < input_size)
    {

        // read byte
        ch = inputByteVector_arg[readPos++];

        // map range
        low += freq[ch] * (range /= freq[256]);
```

```
        range *= freq[ch + 1] - freq[ch];

        // check range limits
        while ((low ^ (low + range)) < top || ((range < bottom) && ((range = -int(low) & (bottom
- 1)), 1)))
        {
            out = static_cast<char> (low >> 24);
            range <<= 8;
            low <<= 8;
            outputCharVector_.push_back(out);
        }

        // update frequency table
        for (j = ch + 1; j < 257; j++)
            freq[j]++;

        // detect overflow
        if (freq[256] >= maxRange)
        {
            // rescale
            for (f = 1; f <= 256; f++)
            {
                freq[f] /= 2;
                if (freq[f] <= freq[f - 1])
                    freq[f] = freq[f - 1] + 1;
            }
        }

    }

    // flush remaining data
    for (i = 0; i < 4; i++)
    {
        out = static_cast<char> (low >> 24);
        outputCharVector_.push_back(out);
        low <<= 8;
    }

    // write to stream
    outputByteStream_arg.write(&outputCharVector_[0], outputCharVector_.size());

    return (static_cast<unsigned long> (outputCharVector_.size()));
}
```

```
unsigned long AdaptiveArithmeticCoder::decodeStreamToCharVector(std::istream &
inputByteStream_arg, std::vector<char>& outputByteVector_arg)
{
	uint8_t ch;
	DWord freq[257];
	unsigned int i, j, f;

	// define limits
	const DWord top = static_cast<DWord> (1) << 24;
	const DWord bottom = static_cast<DWord> (1) << 16;
	const DWord maxRange = static_cast<DWord> (1) << 16;

	DWord low, range;
	DWord code;

	unsigned int outputBufPos;
	unsigned int output_size = static_cast<unsigned> (outputByteVector_arg.size());

	unsigned long streamByteCount;

	streamByteCount = 0;

	outputBufPos = 0;

	code = 0;
	low = 0;
	range = static_cast<DWord> (-1);

	// init decoding
	for (i = 0; i < 4; i++)
	{
		inputByteStream_arg.read(reinterpret_cast<char*> (&ch), sizeof(char));
		streamByteCount += sizeof(char);
		code = (code << 8) | ch;
	}

	// init cumulative frequency table
	for (i = 0; i <= 256; i++)
		freq[i] = i;

	// decoding loop
	for (i = 0; i < output_size; i++)
	{
		uint8_t symbol = 0;
```

```
uint8_t sSize = 256 / 2;

// map code to range
DWord count = (code - low) / (range /= freq[256]);

// find corresponding symbol
while (sSize > 0)
{
    if (freq[symbol + sSize] <= count)
    {
        symbol = static_cast<uint8_t> (symbol + sSize);
    }
    sSize /= 2;
}

// output symbol
outputByteVector_arg[outputBufPos++] = symbol;

// update range limits
low += freq[symbol] * range;
range *= freq[symbol + 1] - freq[symbol];

// decode range limits
while ((low ^ (low + range)) < top || ((range < bottom) && ((range = -int(low) & (bottom
- 1)), 1)))
{
    inputByteStream_arg.read(reinterpret_cast<char*> (&ch), sizeof(char));
    streamByteCount += sizeof(char);
    code = code << 8 | ch;
    range <<= 8;
    low <<= 8;
}

// update cumulative frequency table
for (j = symbol + 1; j < 257; j++)
    freq[j]++;

// detect overflow
if (freq[256] >= maxRange)
{
    // rescale
    for (f = 1; f <= 256; f++)
    {
        freq[f] /= 2;
```

```
                if (freq[f] <= freq[f - 1])
                    freq[f] = freq[f - 1] + 1;
            }
        }
    }

    return (streamByteCount);
}
```

相关代码：
说明：该代码效率很低，因为是将每一个概率都转化为 BigNums 类型进行操作运算；其实可以四个符号或者几个符号为一组，用基本数据类型计算，之后转化为 BigNums 类型，有利于节省资源，提高效率。当然，如果你是分组编码的，不存在位数问题，只是单纯的计算而已

总结

算术编码的优势
与哈夫曼编码类似，算术编码也是一种非定长编码，但其克服了哈夫曼编码每个字符的编码只能为整数的缺陷，其平均字符编码长度更符合频率模型，因此具有更高的压缩比。