《数字媒体技术基础》结课报告：第二部分

郑翰浓，2101212849，信息工程学院

干皓丞，2101212850，信息工程学院

1 选题背景

考虑到我组对熵编码的兴趣，以及ITM15.0代码不便于阅读理解，我组另做了一个熵编码工具，来理解熵编码的过程以及细节。

2 技术实现

2.1 基础功能

该熵编码工具采用的是基于上下文的自适应二值算术编码（CABAC），用于实现文件内容的压缩。编码器的输入为文件内容本身的每个二进制位，从文件的最低比特到最高比特（即小端模式）；输出的编码结果也为二进制串，也按照小端模式写入文件中。解码器的工作与编码器相反，可无损还原出原始文件。

算术编码采用的是增广的整数编码，即用64位无符号整数（uint64_t）来表征编码区间，并通过E1、E2、E3变换来更新区间从而保证编码精度。

2.2 核心：概率模型

概率模型的构建与维护是决定编码性能的核心。显然，由于解码端需与编码端同步更新概率模型，因此概率模型的更新必须只能参照已解码的信息。

2.2.1 条件概率模型

这里，我设计的方案是条件概率，预先指定参数`COR_BIT`，对于文件的第`i`个比特，考虑其前`COR_BIT`比特（滑动窗口）作为条件的概率，若`i ≤ COR_BIT`则需在前面填充0。例如，第`i`比特为1，前`COR_BIT`（=4）比特为0010，则记为条件概率`P(1 | 0010)`，以此类推。

2.2.2 概率模型的更新

这里我们暂且跳过概率模型的初始化，先讲概率模型的更新。每次更新概率模型时，只更新对应场景下的条件概率，将另一边的概率乘以一个小于0的固定参数`RATE`。例如第`i`比特为1，滑动窗口为0010，编码或解码至此，则将`P(0 | 0010)`的值乘以`RATE`（比如`RATE`=0.9），另一边`P(1 | 0010)`相应调大，使得`P(0 | 0010) + P(1 | 0010)`恒等于1。

另外，为了避免极端情况的出现使得概率模型失衡，甚至可能超越最小精度，这里设置了一个保护参数`MIN_PROB`，它限制了每个概率参数可能达到的最小数值。

概率模型的更新部分，C代码实现如下，其中window变量为滑动窗口。

```
const int COR_BIT = 26; /// 相关比特数，取值[0, 27]，如果内存够大理论上无上限。该数值并非越大越好，看数据的关联性
double RATE = .7; /// 概率模型学习率，取值[0.01, 1]，一般取接近1的值
double MIN_PROB = .00001; /// 最小概率限制，取值(0, 0.5)，一般取接近0的值但太小时会有精度问题，大文件会出问题

void update_prob(int bit) {
    prob[window] *= RATE;
    prob[window] += bit ? (MIN_PROB * (1 - RATE)) : ((1 - MIN_PROB) * (1 - RATE));
    window <<= 1;
    window |= bit;
    window &= (1 << COR_BIT) - 1;
}
```

2.2.2 概率模型初始化

初始化概率模型时，简单将所有概率全部设置为0.5即可。其实在这里也本可以没有强行规定，只需要编码端与解码端的协议一致即可正常编解码，只不过接近0.5的初始数值的性能更好。

其实在这一点上可以设计一种加密算法了：基于一个种子（seed）来生成初始概率模型，只有编码端与解码端种子一致时，才能解码成功。该加密方式由于随机数的生成与原始文件内容无法同时被破解，因此可以避免已知明文破解。该算法如下，其中当`seed = 0`时采用全部初始为0.5的常规概率模型。

```
void init() {
    if (seed != 0) {
        srand(seed);
        for (int i = 0; i < (1 << COR_BIT); i++) {
            prob[i] = .5 + .01 * sin(rand());
        }
    } else {
        for (int i = 0; i < (1 << COR_BIT); i++) {
            prob[i] = .5;
        }
    }
    /// 其它代码......
}
```

2.3 踩坑

由于是完全自己写的代码，所以也猜了不少坑，可提醒大家注意的。虽然大家可能不会写这样的底层代码，但也有助于大家理解算术编码的过程。

2.3.1 E1、E2与E3变换

算术编码过程中，必须一直检测E1、E2、E3变换，只要任何时候有一种变换是可行的，那么就必须变换，且编解码端同步变换才能保证解码内容正确。这里面不应该考虑优化或偷懒，例如，若先检测E3，没法E3了再检测E1，没法E1了再E2，没法E2了就结束，这种方法是大错特错的，举个反例，若初始区间为[0.2, 0.3)，它本身不能E3变换但是经过E1后变为[0.4, 0.6)就可以E3了，于是用此方法便少了最终这一步E3变换，导致出错。

因此为了避免出错，最好不要偷懒，E1、E2、E3必须同时检测，只要区间还能变换，就要把三种变换都再检测一遍，直到区间没法进行任何变换。

2.3.2 代码的逻辑规范问题

最初写编码端代码时，为了单方面写代码方便，读文件采用小端模式而写文件采用大端模式，结果导致写解码端代码时特别麻烦，因为读文件是大端模式而写文件是小端模式了，就无法沿用编码端的代码。后来我把编码端的逻辑改了一下，使得读写文件均为小端模式，就消除了这个问题。

3 实验结果

用该工具压缩文件，纯文本能达到很好的压缩率，因其中有大量重复字符，其压缩率能达到接近50%。将部分文件压缩率列表如下：

文件	特征	原始大小（字节）	压缩后大小（字节）	压缩率
日记	markdown纯文本，中文	32,828	21,692	33.9%
日记	pdf，由中文日记导出	548,327	531,581	3.1%
课程报告	markdown纯文本，中文	20,328	14,274	29.8%
main	C代码文件	5,925	3,169	46.5%
测试input	只包含26个大写字母	2,291	529	76.9%

对于图像、视频类文件，由于大多数原始文件如PNG、JPEG、MP4格式已经是经过压缩的，因此再次压缩的效果不理想，大多数压缩率都无法超过10%，甚至有相当一部分为负压缩率，在此不列表了。

4 总结

熵编码对于总字符种类比较少的纯文本文件的压缩效率很好，且我想到的基于初始化概率模型参数的加密方法也非常成功，虽说该方法也许在上个世纪就已有人提出了，但自己研究的体验肯定是不一样的，研究过程中也踩了很多坑，也收获很大。

该熵编码工具为C/C++编写，代码纯手打且纯自己设计实现，见DigitalMediaTech/main.cpp at main · 774889315/DigitalMediaTech (github.com)