基于分块因子的二进制算术编码图像压缩系统

汇报小组：

时间：

2025

目录

CONTENTS

Part 01

一、编码原理、对比及其应用

POWERPOINT DESIGN

算数码编码原理

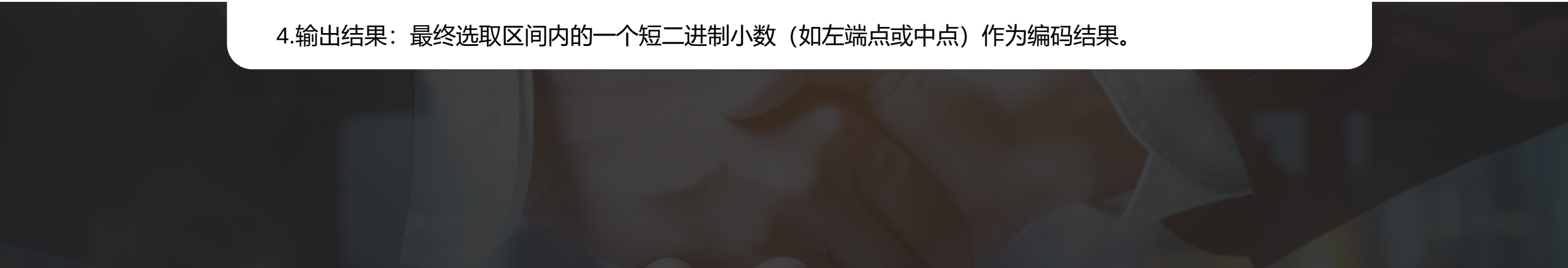
算术码是一种基于概率模型的熵编码方法，通过将整个输入消息映射到[0, 1)区间内的一个实数子区间来实现压缩。具体步骤如下：

1.初始化：初始区间为[0, 1)。

2.迭代细分：根据符号的概率分布，将当前区间划分为若干子区间，每个子区间对应一个符号。

3.区间更新：每输入一个符号，选择对应的子区间作为新的当前区间，并递归细分。

4.输出结果：最终选取区间内的一个短二进制小数（如左端点或中点）作为编码结果。



算数码编码实例

假设信源信号有{A, B, C, D}四个，他们的概率分别为{0.1， 0.4， 0.2， 0.3}，如果我们要对CADACDB这个信号进行编码，那么应该怎样进行呢?

解：首先，我们做好准备工作，按照信源信号的频率为将[0, 1)分解成为若干区间。那么这一步骤完成之后，我们得到以下的表格：

symbol	A	B	C	D
probability	0.1	0.4	0.2	0.3
initial Interval	[0, 0.1)	[0.1, 0.5)	[0.5, 0.7)	[0.7, 1)

准备工作完成之后，我们便可以开始进行编码了。

那么我们首先读入信号：C——因为C在最初始的间隔中是[0.5, 0.7)，所以读入C之后我们的编码间隔就变成[0.5, 0.7)了；

紧接着，我们读入的是A，A在初始区间内是占整个区间的前10%，因此对应这个上来也是需要占这个编码间隔的前10%，因此编码区间变为：[0.5, 0.52)了；

再然后是D，因为D占整个区间的70% ~ 100%，所以也是占用这个编码区间的70% ~ 100%，操作后的编码区间为[0.514, 0.52)

.....

直到最后将信号量全部读出。

最后，我们将这个操作过程绘制成为一张表：

步骤	输入符号	编码间隔	间隔
1	C	[0.5, 0.7)	0.2
2	A	[0.5, 0.52)	0.2*0.1
3	D	[0.514, 0.52)	0.2*0.1*0.3
4	A	[0.514, 0.5146)	0.2*0.1*0.3*0.1
5	C	[0.5143, 0.51442)	0.2*0.1*0.3*0.1*0.2
6	D	[0.514384, 0.51442)	0.2*0.1*0.3*0.1*0.2*0.3
7	B	[0.5143876, 0.514402)	0.2*0.1*0.3*0.1*0.2*0.3*0.4
8	最后从得到的编码区间中[0.5143876, 0.514402)任取一个数就可以作为编码而输出：0.5143876		

算数码特点

接近理论极限：能够以分数位表示符号，压缩率高于霍夫曼码/香农码。

动态概率适应：支持实时更新概率模型，适合非平稳数据流。

长消息高效性：整条消息编码为一个数，避免逐符号的整数位浪费。

优点

计算复杂度高：高精度浮点运算对硬件要求较高。

错误敏感：传输错误可能导致后续解码完全失效。

实现复杂：需处理区间精度衰减。

局限性

算数码与香农码对比

算数码

码长分配：分数位，接近理论最优

压缩效率：更高，尤其对非均匀分布

实现复杂度：高（需浮点运算与区间管理）

适应性：支持动态概率模型

典型应用：多媒体压缩（如CABAC）

香农码

码长分配：整数位（$\geqslant -\log_2(p)$的最小整数）

压缩效率：较低，存在整数位冗余

实现复杂度：低（基于符号概率排序分配码字）

适应性：静态概率模型

典型应用：理论场景，实际应用较少

算数码典型应用

典型应用

领域	技术案例	数码核心作用	性能指标
计算机科学	哈夫曼编码	数据压缩优化	压缩率提升30%
通信工程	5G OFDM调制	高频谱效率传输	峰值速率20Gbps
数据加密	AES-256加密	高安全性数据保护	密钥空间2^256
人工智能	神经网络量化	模型参数8位整数量化	推理速度提升4倍，精度损失<1%

压缩工具及其算法

7-Zip：

LZMA/LZMA2算法：用于对文件进行高压缩比的压缩与解压缩，常用于整理和分发大量文件资料，如软件安装包、文档资料、图片等。

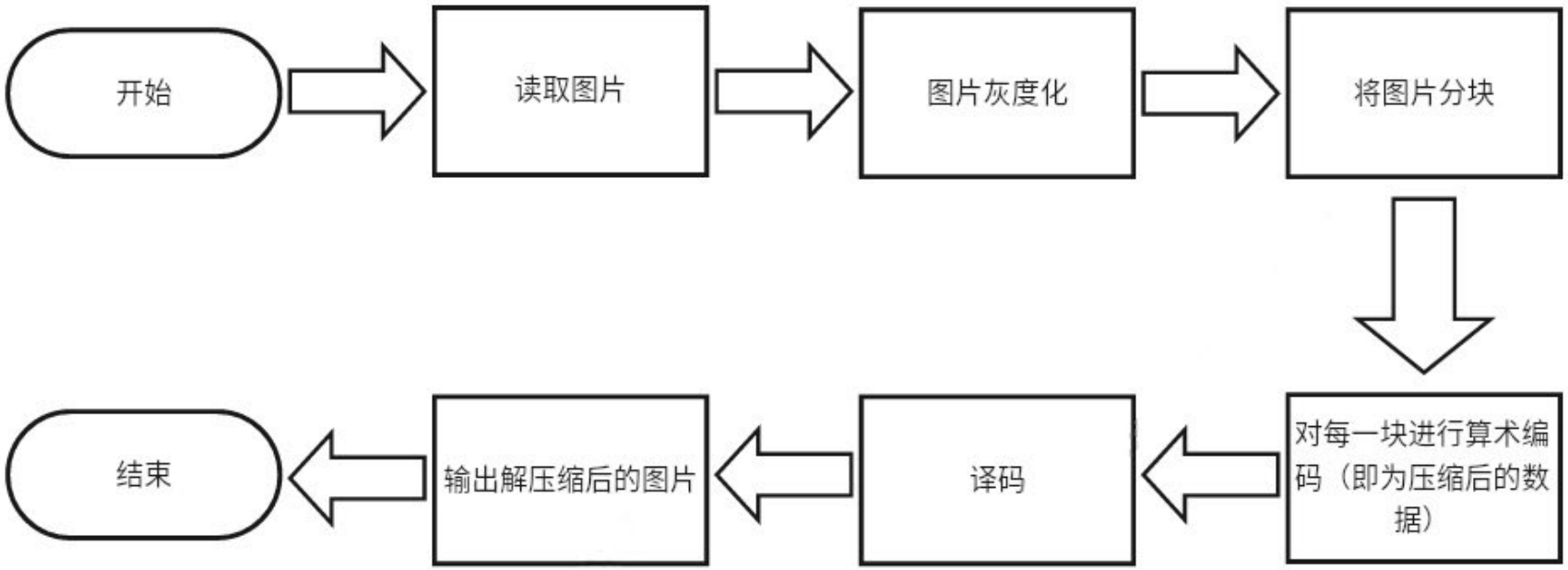
PPMD算法：对文本等数据进行压缩时效果良好，适用于文本内容较多的文件的压缩存储与传输。

Part 02

二、系统程序流程与运行结果

POWERPOINT DESIGN

程序流程



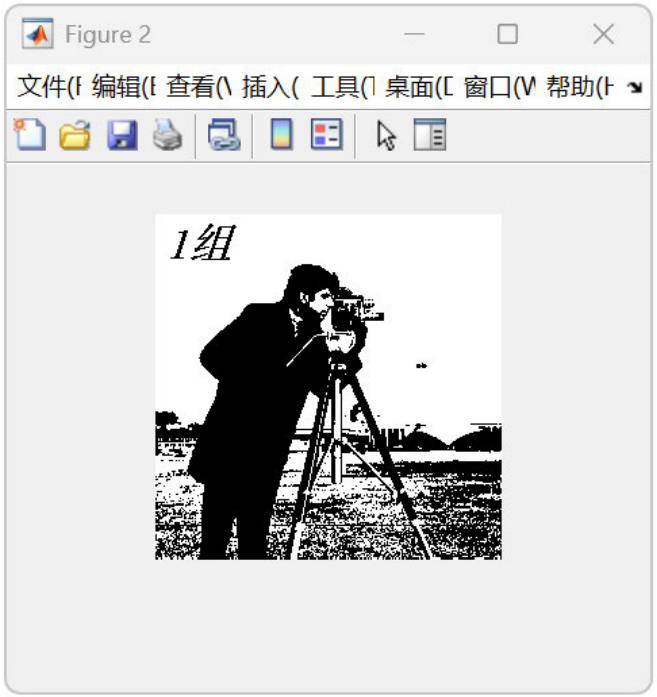
图片读取+灰度化

读取图片

```
%读取图片数据
I=imread('图片1.bmp');
figure,imshow(I);
```

BMP图片灰度化

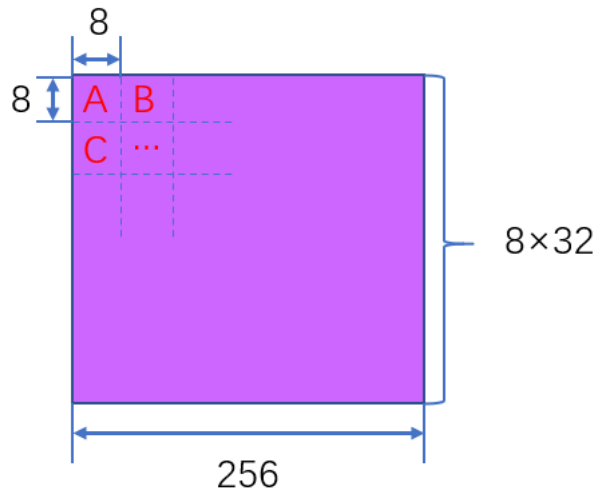
```
%将图片变为一通道
J=rgb2gray(I);
figure,imshow(J);
imwrite(J,'1组-灰度图.bmp');
```



将图片分块

```
%将图片进行分块
k=8;
m=256/k;
A=zeros(m,m,k,k);
A2=zeros(256,256);
B={};
C={};
cnt=0;
```

进行如上操作后，我们将整个由256×256个像素组成的灰度图，分为32×32个包含8×8像素的块，如下图所示:



将图片分块

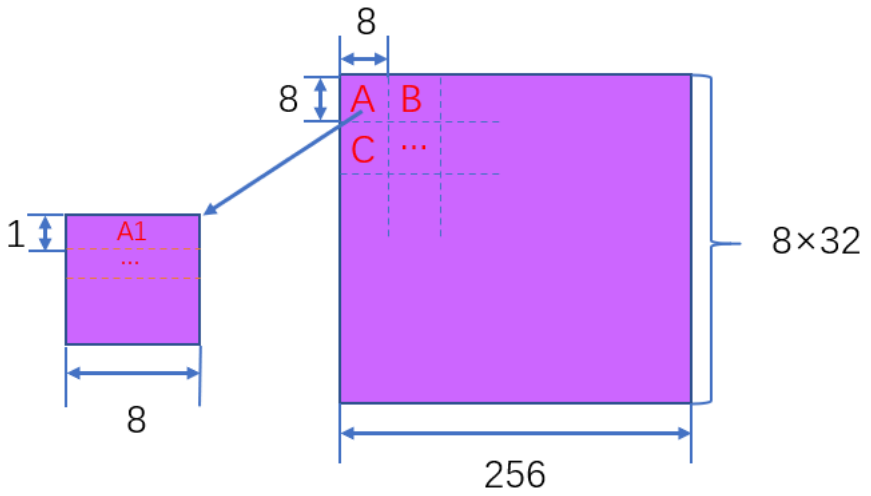
我们发现，已给的图片只有0和255两个灰度，意味着本图片只有纯黑和纯白，没有灰色。
这里展示1-32行、1-32列的灰度数据:

255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	0	255	0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	0	255	255	0	0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	0	0	0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	0	0	0	0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	0	0	255	255	255	0	255	0	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	0	0	0	0	0	0	0	255	255	0	0	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	0	0	0	0	0	0	0	0	0	255	0	255	0	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	0	0	0	0	0	0	0	255	0	0	255	0	0	0	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	0	0	0	0	0	0	0	0	0	255	0	255	255	0	0	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	0	0	0	0	0	0	0	0	255	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	0	255	255	255	255	255	255
255	255	255	255	255	255	0	0	0	0	0	0	0	0	0	0	255	0	0	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	0	0	0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	0	0	0	0	0	0	0	0	255	255	0	255	255	0	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	0	0	0	0	0	0	0	0	0	255	0	255	255	255	0	255	255	255	255	255	255	255	255	255	255	255	255
0	0	255	255	255	0	0	0	0	0	0	0	0	0	255	0	255	255	0	255	0	255	255	255	255	255	255	255	255	255	255	255
0	0	0	0	0	0	0	0	0	0	0	0	0	255	255	0	255	255	0	255	0	0	0	255	0	255	255	255	255	255	0	255
0	0	0	0	0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	0	0	0	0	255	0	0	0	0
0	255	0	255	0	0	0	0	0	0	0	0	0	0	0	0	255	255	255	0	255	0	255	255	255	255	255	255	255	255	255	255
255	255	255	255	0	0	0	0	0	0	0	0	0	255	0	255	255	255	255	255	255	255	255	0	0	0	0	0	0	0	0	0
255	255	0	0	0	0	0	0	0	0	0	0	255	255	0	255	255	255	255	255	0	0	0	0	0	255	0	0	0	0	0	0
0	255	255	255	255	0	0	0	0	0	0	0	0	0	0	0	0	255	255	255	255	0	0	255	255	255	255	255	255	0	0	0
255	255	255	255	255	0	0	0	0	0	0	0	255	255	0	255	255	255	255	255	0	255	255	0	255	0	0	0	0	0	255	255
0	0	255	255	255	0	0	0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	255	0	0	0	0	0	0	0	0	0
255	255	0	255	255	0	0	0	0	0	0	255	0	0	0	0	0	0	0	0	0	255	0	0	0	0	255	0	0	0	0	0
0	0	0	0	255	0	0	0	0	0	0	0	255	255	255	0	255	0	0	0	255	255	255	0	0	0	0	0	0	0	255	255
0	0	0	0	0	0	0	255	0	0	0	255	255	255	0	0	255	0	0	0	0	255	255	0	255	0	255	255	0	255	0	0
255	255	0	0	255	0	0	255	0	0	0	0	255	255	255	0	255	0	0	0	0	255	255	255	255	255	0	255	0	0	0	255

对每一块分别进行算术编码

代码中encode函数为自己编写的算术编码子函数，因为代码过于冗长，不在这里做展示。
具体算法是将每块的每个像素的灰度数据进行8位二进制转换，而后将8×8个灰度数据按照行进行算术编码，得到每个块每行的算术编码，即右图中的A1等（其中A1等为某个二进制数）。

```
% 将矩阵进行k*k分块编码，对分的每一块按行进行算术编码
for i=1:m
    for j=1:m
        A(i,j,:,:)=J(1+k*(i-1):k*i,1+k*(j-1):k*j);
        A0=J(1+k*(i-1):k*i,1+k*(j-1):k*j);
        % 调试语句
        % kk=(i-1)*m+j;
        % if kk==35
        %     debug=0;
        % end
        [B0,C0,cnt0]=encode(A0,k);
        B=[B;B0];
        C=[C;C0];
        cnt=cnt+cnt0;
    end
```



输出压缩文件并计算压缩比

因为整个图片的算术编码矩阵过大，这里只列出1-4列、1-32行的算术编码

```
val(:,:,1) =

  列 1 至 4

    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[   0 1 1 1 1 1 1]}    {[               1]}
    {[               1]}    {[               1]}    {[ 0 1 1 0 0 0 1 ... ]}    {[ 0 1 1 1 1 1 1 ... ]}
    {[               1]}    {[ 0 1 1 1 1 1 1 ... ]}    {[ 0 1 0 1 0 1 0 ... ]}    {[ 0 1 1 1 1 1 1 ... ]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               1]}    {[               1]}    {[               1]}    {[               1]}
    {[               0]}    {[               0]}    {[               1]}    {[               1]}
    {[               0]}    {[               0]}    {[               0]}    {[               0]}
    {[               0]}    {[               0]}    {[       0 0 0 0 1]}    {[  0 0 1 1 0 0 1 1]}
    {[ 0 0 0 1 1 0 1 1]}    {[ 0 1 1 1 1 1 1 ... ]}    {[  0 0 0 1 0 1 0 1]}    {[0 1 0 1 0 0 0 0 1]}
    {[               1]}    {[               1]}    {[               1]}    {[ 0 0 0 0 0 1 0 ... ]}
    {[0 1 1 1 1 1 1 ... ]}    {[ 0 1 1 1 0 1 0 ... ]}    {[       0 0 0 1 1]}    {[               0]}
    {[0 1 0 1 1 1 1 ... ]}    {[               1]}    {[               1]}    {[ 0 1 1 1 1 1 1 ... ]}
    {[0 1 1 0 1 1 1 ... ]}    {[               1]}    {[               1]}    {[ 0 1 1 1 1 1 1 ... ]}
    {[0 0 0 0 0 0 0 ... ]}    {[      0 0 0 0 0 1]}    {[ 0 1 1 1 1 1 1 ... ]}    {[               1]}
    {[     0 1 0 1 0 1]}    {[0 1 1 0 0 1 1 1 1]}    {[      0 0 0 0 0 1]}    {[  0 1 0 1 0 1 0 1]}
    {[   0 0 0 0 0 0 1]}    {[               0]}    {[ 0 0 0 0 0 0 1 ... ]}    {[0 0 1 1 1 0 0 1 1]}
    {[0 0 0 0 1 0 1 ... ]}    {[0 0 1 1 0 0 0 0 1]}    {[               0]}    {[ 0 0 0 0 0 0 0 0 ... ]}
    {[0 0 1 0 1 1 1 ... ]}    {[ 0 0 1 1 1 1 1 1 ... ]}    {[0 0 0 0 1 0 0 0 1]}    {[    0 0 1 1 0 0 1]}
```

```
% 计算压缩比
y=(256*256*8)/cnt;
fprintf('压缩比为: %.2f%/n', y);
disp(' ');
% 将得到的二进制算术编码写入文件
% 其中Zip(i,j,:)为原图像矩阵中第i行第j列的分块矩阵的按行二进制算术编码
for i=1:m
    for j=1:m
        D=C(k*((i-1)*m+j-1)+1:k*((i-1)*m+j-1)+k);
        Zip(i,j,:)=D;
    end
end
save('二进制算术编码.mat', 'Zip');
disp('二进制算术编码已写入文件"二进制算术编码.mat"');
```

通过循环迭代，可以算出压缩前数据长度为256×256×8，压缩后数据长度为 35568 ，压缩比Y=14.74。

译码并计算失真率

```
% 分块译码
% 对算术编码进行译码，A2为译码后的图像矩阵
for i=1:m
    for j=1:m
        % 调试语句
        % kk=(i-1)*m+j;
        % if kk==35
        %     debug=0;
        % end
        A0=J(1+k*(i-1):k*i,1+k*(j-1):k*j);
        x=B(k*((i-1)*m+j-1)+1:k*((i-1)*m+j-1)+k);
        A1=decode(A0,k,x);
        A2(1+k*(i-1):k*i,1+k*(j-1):k*j)=A1;
    end
end
% 计算失真率
A2=uint8(A2);
yl=length(find(A2-J))/256*256;
fprintf('失真率为: %.2f%%\n', yl * 100);
```

译码过程即为编码过程的逆过程（即为代码中的decode函数），将编码流程反向进行，便能得到最后的解压缩图像。

其中，计算失真率YL=0.00%,说明此编码方式的确为无失真编码。

图片对比

下图为原图片与解压缩后图片的对比，可以看到，解压后图片与原图片完全一样，但解压缩后图片大小远远小于原图片大小，且从24位变为8位，是因为经过了单通道处理。

Part 03

三、算法特点——分块因子法

POWERPOINT DESIGN

分块因子算法

01

遇到问题

在算术编码过程中，逼近出的小于1的数位数很多，导致计算机在编程当中会将后几位数字截断，使编码失真，无法实现无失真压缩图像

02

解决办法

在每次迭代概率区间时，将区间界乘某个倍率因子，将概率区间变为整数区间，从而避免小数位数太多引起的截断误差

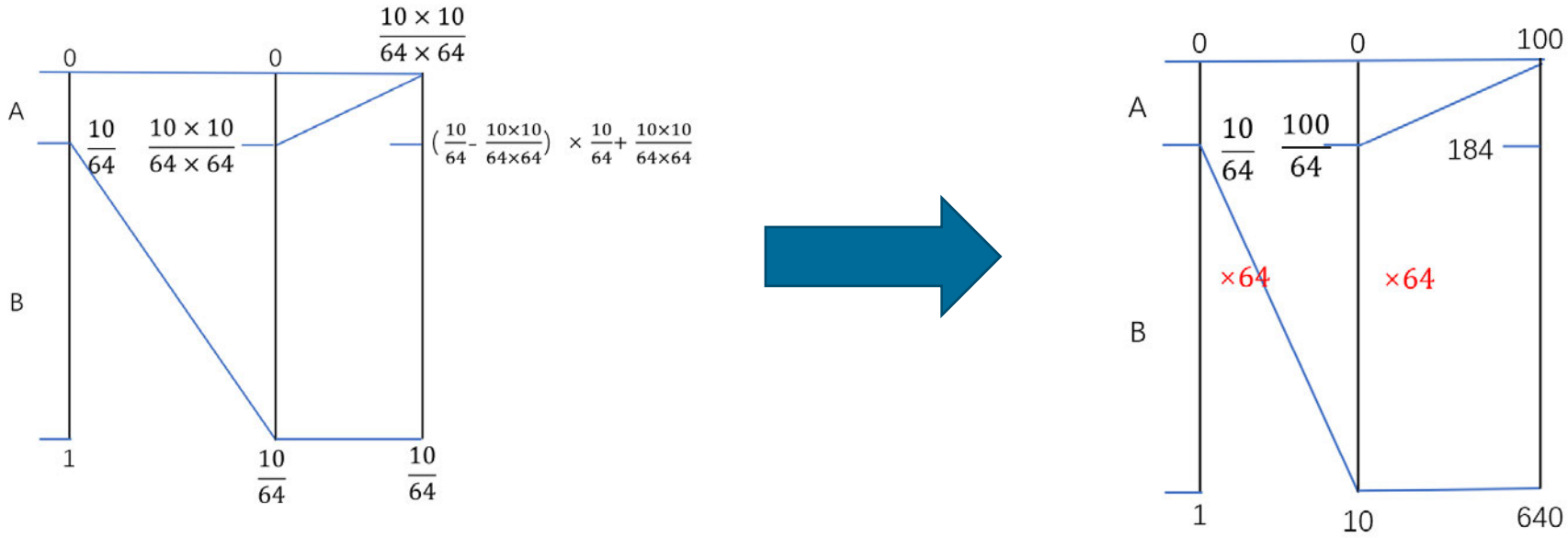
03

效果

通过编程实践，能够使计算机保留存储全部位数，从而转化为准确的二进制算数编码，保证无失真压缩

分块因子算法实例

在算术编码分配概率区间时，因为分块方式为8×8像素分为一块，每个像素用8位二进制数表示，所以每一行有64个二进制字符，所得概率分布必定为0-$\frac{x}{64}$-1。我们考虑到每次计算概率区间时都乘一行的二进制字符数（即64），会将每次迭代的概率区间都转化为整数，不会出现小数，计算机便不会差生截断误差，如下图所示：



分块因子算法实例

当计算到最后一位，即第64位，我们得到一个整数区间，但关键是如何取区间内的某一个数，我们考虑到将区间界转化为二进制数，将两个二进制数从左到右一样的数保留，第一位不一样的数取成1，后面补零，得到最后一步估计出的整二进制数，将其小数点向前移动6×64位（因为每次对概率区间乘64，对二进制数来说是小数点向后移动6位）

Part 04

四、总结

POWERPOINT DESIGN

总结

压缩文件

本小组通过分块算术编码，实现了压缩图片的数字化输出

压缩比

通过有效的分块压缩，我们最终得到：压缩前数据长度为256×256×8，压缩后数据长度为35568，压缩比Y=14.74

文件对比

输出解压缩文件后，将原图片与处理后图片对比，观察到图片无失真

Part 05

五、分工与参考文献

POWERPOINT DESIGN

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

参考文献

[1] Rissanen J., Langdon G. G. Arithmetic coding[J]. IBM Journal of Research and Development, 1979, 23(2): 149-162.

[2] Witten I. H., Neal R. M., Cleary J. G. Arithmetic coding for data compression[J]. Communications of the ACM, 1987, 30(6): 520-540.

[3] Marpe D., Schwarz H., Wiegand T. Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2003, 13(7): 620-636.

[4]Joshi R L, Fischer T R, Marcellin M W, et al. Arithmetic and trellis coded quantization[C]//Proceedings of the IEEE, International Symposium on Information Theory. IEEE, 1994: 233.

[5]刘晓晓.高效预处理和混合熵编码的图像无损压缩算法研究[D].上海大学,2023.DOI:10.27300/d.cnki.gshau.2023.000514.

[6]吴晓云.算术编码算法在图像压缩中的研究[J].计算机与数字工程,2017,45(09):1863-1865.

[7]赵耀, 雷玉堂. 基于算术编码的无损图像压缩算法[C]. 中国图象图形学学会第十届全国图象图形学学术会议论文集, 2001: 434-437.

谢谢大家

汇报小组：

时间：

2025