

Data structure实验报告

—基于Huffman编码的文件压缩/解压工具

PB21050988杜朋澈

实验目的

在合适条件下利用Huffman编码对文件进行压缩可以减少其占用的存储空间, 本次实验将实现一个基于Huffman编码的文件压缩软件。

实现功能

- 基础功能
 - 对任意类型的文件进行以单字节为数据单元的Huffman编码的压缩和解压缩操作
 - 程序运行后由终端用户交互指定程序执行的确切任务
- 附加功能
 - 可指定的数据单元长度
 - 多叉树Huffman编码改进

因多叉树Huffman编码只可能在叉数为 2^k ($k \in \mathbb{Z}$)时才可能具备更高的压缩效率, 故在这里仅展示四叉树结构。

模块简介

- 基础结构

二叉Huffman节点结构:

```

1 typedef struct{
2     unsigned int byte; //数据单元内容
3     long long weight; //节点权重
4     int parent,lchild,rchild; //树节点基本信息
5     unsigned char *code; //数据单元Huffman编码
6     unsigned int codelength; //数据单元Huffman编码长度
7 }HuffNode,*HuffTree;

```

这是基础二叉树Huffman编码的Huffman树节点结构，其中：

- byte域大小为4 bytes，即最多可指定的数据单元长度为4 bytes，只有叶子节点的byte域有效。
- weight域为节点权重。
- parent, lchild, rchild分别为树的父母节点和左右孩子节点信息。
- code域为一个长度不固定的字符串，若对应节点为叶子节点，则以串形式存储其Huffman编码。
- codelength域存储Huffman编码长度。

二叉Huffman树结构：

```

1 HuffTree T=(HuffTree)malloc((2*k-
  1)*sizeof(HuffNode));

```

根据Huffman编码的原则，Huffman树应为正则二叉树，若其叶节点数目为 k ，则其总结点数为 $2k - 1$ 。本实验中的Huffman树为包括 $2k - 1$ 个HuffNode的数组，其重要特征是只有前 k 个节点为叶子节点，其余均为中间节点，这样便可通过数组下标快速访问任意内容数据单元的节点。

四叉Huffman节点结构：

```

1 typedef struct{
2     unsigned int byte;
3     long long weight;
4     int parent,achild,bchild,cchild,dchild;
5     unsigned char *code;
6     unsigned int codelength;
7 }HuffNode_Quad,*HuffTree_Quad;

```

这是改进四叉树Huffman编码的Huffman树节点结构，其中与二叉树的不同点在于：

- achild, bchild, cchild, dchild共四个孩子节点。

四叉Huffman树结构：

```

1 HuffTree_Quad T=(HuffTree_Quad)malloc(((4*k-
2 1)/3+1)*sizeof(HuffNode_Quad));

```

四叉Huffman树未必是正则的，但如果其叶节点数目为 k ，则其总结点数不会超过 $\frac{4k-1}{3} + 1$ 。其余特征与二叉Huffman树类似。

• 辅助结构

优先队列：

```

1 #include <queue>
2 typedef pair<long long,unsigned int>Node;
3 priority_queue<Node,vector<Node>,greater<Node>>
  heap;

```

pair Node类型用于存储任意节点的主要信息：first域为节点权重，second域为节点数据单元内容。

利用头文件queue中的类 priority_queue 依照权重大小进行小顶堆排序。

• 主要功能模块

二叉Huffman树压缩模块：

函数原型

```
1 void Compress_Bin(char* I_FILE_NAME, char*  
  O_FILE_NAME, short bymod);
```

参数简介：

- I_FILE_NAME：压缩操作的源文件路径。
- O_FILE_NAME：压缩操作的目标文件路径。
- bymod：压缩操作的数据单元长度。

功能简介：

读取源文件，获取数据单元频率表，建立Huffman树并依此对数据单元进行编码，最终向目标文件写入压缩后的编码内容。

重要参数

```
1 int unitnum=0; //源文件数据单元数  
2 int nodenum=0; //Huffman树有效节点数  
3 int amplnum=0; //源文件补充字节数
```

- 有效节点：源文件中未必会出现所有不同的数据单元，为提高效率，只记权重大于0的叶子节点为有效节点。
- 补充字节：当数据单元长度不为1 byte时，源文件的大小可能并不是数据单元长度的整数倍，这个时候需要将源文件补充至合适的大小（并非真正写入额外内容，只是作为重要参数进行记录和提示）。

数据单位读取与统计

```
1 while(!feof(fp)){  
2     unsigned int reg=0; //数据单位寄存器  
3     int pos=0; //指示寄存器当前读取位置（最大为bymod）  
4     for(int i=0; i<2*bymod; i++){ //以0.5字节为粒度  
5         if((i+1)%2==0){  
6             unsigned char buf=0; //缓冲区  
7             fread(&buf, 1, 1, fp);
```

```

8         if(feof(fp)){
9             break;
10        }
11        reg=reg<<8;
12        reg=reg+buf;
13        pos++;
14    }
15    } //—读取模块—
16    if(feof(fp)){
17        amplnum=(bymod-pos)%bymod;
18        if(!amplnum)break;
19    }
20    T[reg].weight++;
21    unitnum++;
22 }
23 fclose(fp);
24 priority_queue<Node,vector<Node>,greater<Node>>
    heap;
25 for(unsigned int i=0;i<k;i++){
26     if(T[i].weight>0){
27         heap.push({T[i].weight,i});
28         nodenum++;
29     }
30 }

```

- 读取模块是以0.5 bytes为粒度的。计数器i每满足1 byte的需求，就从源文件中读取一1 byte内容至缓冲区中并写入寄存器reg，循环结束时，reg中存储一个数据单元长度的内容。
- 读取到一个数据单元后通过下标立刻查询对应节点，对应权重与参数unitnum均增加。
- 节点内容统计完成后将有效节点推入小顶堆中进行排序，便于后续建树操作，同时记录有效节点数。

构建Huffman树

```

1 unsigned int c;
2 for(c=k;heap.size()>1;c++){
3     Node temp=heap.top();
4     heap.pop();

```

```

5      long long curL_W=temp.first;
6      unsigned int curL_B=temp.second;
7      temp=heap.top();
8      long long curR_W=temp.first;
9      unsigned int curR_B=temp.second;
10     heap.pop();
11     T[c].weight=curL_W+curR_W;
12     T[c].lchild=curL_B;
13     T[c].rchild=curR_B;
14     T[curL_B].parent=c;
15     T[curR_B].parent=c;
16     heap.push({T[c].weight,c}); //pair的byte域大于255
    则说明其为中间节点
17 }

```

依照传统Huffman算法执行的建树过程，每次从小顶堆中弹出两个节点，与新的中间节点连接，权重相加，并在T[k]及之后的空间存入中对应的中间节点信息，然后将其入堆，循环至堆中只剩一个节点，也就是根节点。

数据单元编码

```

1  for(unsigned int i=0;i<k;i++){ //遍历叶子节点
2      if(T[i].weight){
3          int j=0;
4          int c=i;
5          for(int p=T[i].parent;p!=-1;p=T[p].parent)
6          {
7              if(T[p].lchild==c){
8                  T[i].code[j]='0';
9              }
10             if(T[p].rchild==c){
11                 T[i].code[j]='1';
12             } //左0右1
13             T[i].codelength++;
14             j++;
15             c=p;
16         }
17         T[i].code[j]='\0';
18     } //—数据单元编码—

```

从叶子节点向根部搜索，按照左0右1构建逆序编码。

源文件头部信息写入

```
1 unsigned char mode=2;
2 fwrite(&mode,1,1,fp_OS);
3 fprintf(fp_OS,"%d,%s%d,%d,%d,%hd",strlen(FILE_EXTEN
  N),FILE_EXTEN,unitnum,nodenum,amplnum,bymod);
4 for(int i=0;i<k;i++){
5     if(T[i].weight){
6
7         fprintf(fp_OS,"%u,%lld",T[i].byte,T[i].weight);
8     }
```

头部信息依次为：

- 压缩数据单元长度
- 源文件后缀名串长
- 源文件后缀名
- 源文件数据单元数
- Huffman树有效节点数
- 源文件补充字节数
- 有效节点的数据单元内容及其对应权重

源文件编码信息写入

```
1 unsigned char buf=0; //缓冲区
2 int ComBytes=0; //压缩后文件字节数
3 int pos=0;
4 while(!feof(fp_IS)){
5     unsigned int reg=0; //数据单位寄存器
6     for(int i=0;i<2*bymod;i++){
7         if((i+1)%2==0){
8             unsigned char buffer=0; //内部缓冲区
9             fread(&buffer,1,1,fp_IS);
10            if(feof(fp_IS)){
```

```

11         break;
12     }
13     reg=reg<<8;
14     reg=reg+buffer;
15 }
16 }//—读取模块—
17 if(feof(fp_IS)){
18     if(!amplnum)break;
19 }
20 for(int i=T[reg].codelength-1;i≥0;i--){
21     switch(T[reg].code[i]){
22         case '1':{
23             buf++;
24             break;
25         }
26         case '0':{
27             break;
28         }
29         default:cout<<"error:写入时发生错误"
<<endl;
30     }
31     pos++; //指示缓冲区下一bit
32     if(pos==8){
33         fwrite(&buf,1,1,fp_OS);
34         pos=0;
35         buf=0;
36         ComBytes++;
37     }//缓冲区满8 bits执行一次输入 | 维护buf和pos
38     buf=buf<<1;
39 }//—文件写入—
40 }
41 while((pos≠7)&&(pos≠0)){
42     buf=buf<<1;
43     pos++;
44 }//不满8 bits补0后输出
45 if(pos≠0)
46     fwrite(&buf,1,1,fp_OS);
47 ComBytes++; //—文件末尾维护—

```


根据读取到的节点的对应编码，向缓冲区中写入对应内容，每满8 bits向目标文件中写入1 byte，并重置位置指示和缓冲区。编码至文件末尾时若出现编码内容总长度不可被8整除的情况（即文件字节数不为整数），则末端补零后再写入一次。

数据统计输出

```
1 cout<<"total bytes: "<<ComBytes<<" Bytes"<<endl;
2 double ratio=(double)ComBytes/(double)
  (bymod*unitnum);
3 cout<<"compression ratio: "<<ratio*100.0<<"%"
  <<endl;
4 cout<<"---compression completed---"<<endl;
```

输出压缩后的总字节数和压缩率。

二叉Huffman树解压模块：

函数原型

```
1 void Decompress_Bin(char* I_FILE_NAME, char* name);
```

参数简介：

- I_FILE_NAME：待解压文件的路径
- name：输出结果文件的名称（无需后缀名）

功能简介：

读取目标文件头部信息，建立Huffman树，再读取目标文件的编码内容，解码后写入解压后的结果文件。

头部信息读取

```

1 fread(&mode,1,1,fp_IS);
2 fscanf(fp_IS,"%d",&length);
3 fread(&extend,length,1,fp_IS);
4 fscanf(fp_IS,"%d,%d,%d,%hd",&unitnum,&nodenum,&am
  plnum,&bymod);
5 char* O_FILE_NAME=strcat(name,extend);
6 for(int i=0;i<nodenum;i++){
7     unsigned int reg;
8     long long w;
9     fscanf(fp_IS,"%u,%lld",&reg,&w);
10    heap.push({w,reg});
11    T[reg].weight=w;
12 }

```

获取目标文件内的全部头部信息，直接建立节点频率表。

构建Huffman树

与前述方法相同，不再赘述。

目标文件解码信息写入

```

1 unsigned char buf; //缓冲区
2 int root=-c; //根节点位置
3 int pos=0;
4 unsigned char step; //掩码结果
5 int cnt=unitnum;
6 unsigned int p=root; //游标
7 fread(&buf,1,1,fp_IS);
8 while(!feof(fp_IS)){
9     while(pos!=8){
10         step=buf&0x80;
11         switch(step){
12             case 0x00:{
13                 p=T[p].lchild;
14                 break;
15             }
16             case 0x80:{
17                 p=T[p].rchild;

```

```

18         break;
19     }
20     default: cout<<"error:解压写入时发生错误"
<<endl;
21     }
22     if((T[p].lchild==-1)&&(T[p].rchild==-1)){
23         cnt--;
24         if(cnt==0){
25             for(int i=0;i<bymod-AMPLNUM;i++){
26                 unsigned int ch=T[p].byte;
27                 unsigned char m=ch>>(8*(bymod-
AMPLNUM-i-1));
28                 fwrite(&m,1,1,fp_OS);
29             }
30             break;
31         }
32         else{
33             for(int i=0;i<bymod;i++){
34                 unsigned int ch=T[p].byte;
35                 unsigned char m=ch>>(8*(bymod-
i-1));
36                 fwrite(&m,1,1,fp_OS);
37             }
38             p=root;
39         }
40     }
41     buf=buf<<1;
42     pos++;
43 }
44 if(cnt==0)break;
45 fread(&buf,1,1,fp_IS);
46 pos=0;
47 }

```

读取编码内容并写入时主要涉及位操作。

每次从目标文件读1 byte至缓冲区，与0x80掩码，获取最高bit的内容，并依此从根节点向叶子节点搜索。若历经叶子节点则写入对应数据单元内容。

设有计数器cnt，以确保在读取目标文件末尾时不会被补充的0位干扰，从而输出正确的数据单元内容和数目。

同时，若计数器cnt显示此时正在输出最后一个数据单元，则进行一次额外判断，根据amplnum的值确定需要忽略掉的补充字节数。

四叉树与二叉树的压缩与解压算法大同小异，故在此只介绍主要改动点。

四叉Huffman树压缩模块：

构建Huffman树

```
1  unsigned int c=k;
2  int cnt=0;
3  long long cur_W;
4  unsigned int cur_B;
5  while(!heap.empty()){
6      Node temp=heap.top();
7      heap.pop();
8      cnt++;
9      cur_W=temp.first;
10     cur_B=temp.second;
11     switch(cnt){
12         case 1:{
13             T[c].achild=cur_B;
14             break;
15         }
16         case 2:{
17             T[c].bchild=cur_B;
18             break;
19         }
20         case 3:{
21             T[c].cchild=cur_B;
22             break;
23         }
24         case 4:{
25             T[c].dchild=cur_B;
26             break;
27         }
28         default:cout<<"建树时发生错误"<<endl;
29     }
30     T[cur_B].parent=c;
```

```

31     if(cnt==4){
32         cnt=0;
        T[c].weight=T[T[c].achild].weight+T[T[c].bchild].w
        eight+T[T[c].cchild].weight+T[T[c].dchild].weight;
33         heap.push({T[c].weight,c});
34         c++;
35     }
36 }
37 switch(cnt){
38     case 1:{
39         T[cur_B].parent=-1;
40         unsigned int root=cur_B;
41         break;
42     }
43     case 2:{
44
45         T[c].weight=T[T[c].achild].weight+T[T[c].bchild].
weight;
46         unsigned int root=c;
47         break;
48     }
49     case 3:{
50
51         T[c].weight=T[T[c].achild].weight+T[T[c].bchild].
weight+T[T[c].cchild].weight;
52         unsigned int root=c;
53     }
54     default:cout<<"建树时发生错误"<<endl;
55 }

```

设计计数器cnt在1~4间循环计数，并依次与新中间节点的a b c d四个孩子域相连接，每次cnt计到4则说明中间节点满载，将权重相加并开辟新节点继续重复上述过程。值得注意的是四叉Huffman树未必为正则树，故在全部节点连接完成后还需要根据计数器可能出现的三种状态进行一次维护（值得一提的是，对于新节点只连接一个孩子的情况我们需要将新节点删除）。

源文件编码信息写入

```

1 unsigned char buf=0; //缓冲区
2 int ComBytes=0; //压缩后文件字节数

```

```

3  int pos=0;
4  while(!feof(fp_IS)){
5      unsigned int reg=0; //数据单位寄存器
6      for(int i=0;i<2*bymod;i++){ //以0.5字节为粒度
7          if((i+1)%2==0){
8              unsigned char buffer=0; //内部缓冲区
9              fread(&buffer,1,1,fp_IS);
10             if(feof(fp_IS)){
11                 break;
12             }
13             reg=reg<<8;
14             reg=reg+buffer;
15         }
16     }
17     if(feof(fp_IS)){
18         if(!ampLnum)break;
19     }
20     for(int i=T[reg].codelength-1;i>=0;i--){
21         switch(T[reg].code[i]){
22             case '0':{
23                 break;
24             }
25             case '1':{
26                 buf=buf+1;
27                 break;
28             }
29             case '2':{
30                 buf=buf+2;
31                 break;
32             }
33             case '3':{
34                 buf=buf+3;
35                 break;
36             }
37             default:cout<<"error:写入时发生错误"
<<endl;
38         }
39         pos=pos+2; //指示下一bit
40         if(pos==8){
41             fwrite(&buf,1,1,fp_OS);

```

```

42         pos=0;
43         buf=0;
44         ComBytes++;
45     } //缓冲区满8 bits执行一次输入 | 维护buf和pos
46     buf=buf<<2;
47 }
48 }
49 while((pos!=6)&&(pos!=0)){
50     buf=buf<<2;
51     pos=pos+2;
52 } //不满8 bits补0后输出
53 if(pos!=0)
54     fwrite(&buf,1,1,fp_OS);

```

四叉树的编码每一位可能出现四种情况：0, 1, 2, 3即对应二进制的00, 01, 10, 11, 因此缓冲区每次读取编码内容时应左移两位，位置指示随之改变。

四叉Huffman树解压模块：

构建Huffman树

与前述方法相同，不再赘述。

目标文件解码信息写入

```

1  unsigned char buf; //缓冲区
2  int pos=0;
3  unsigned char step; //掩码结果
4  int cnt=unitnum;
5  unsigned int p=root; //游标
6  fread(&buf,1,1,fp_IS);
7  while(!feof(fp_IS)){
8      while(pos!=8){
9          step=buf&0xc0;
10         switch(step){
11             case 0x00:{
12                 p=T[p].achild;
13                 break;
14             }

```

```

15         case 0x40:{
16             p=T[p].bchild;
17             break;
18         }
19         case 0x80:{
20             p=T[p].cchild;
21             break;
22         }
23         case 0xc0:{
24             p=T[p].dchild;
25             break;
26         }
27         default:cout<<"error:解压写入时发生错误"
<<endl;
28     }
29     if((T[p].achild==-1)&&(T[p].bchild==-1)&&
(T[p].cchild==-1)&&(T[p].dchild==-1)){
30         cnt--;
31         if(cnt==0){
32             for(int i=0;i<bymod-amplnum;i++){
33                 unsigned int ch=T[p].byte;
34                 unsigned char m=ch>>(8*(bymod-
amplnum-i-1));
35                 fwrite(&m,1,1,fp_OS);
36             }
37             break;
38         }
39         else{
40             for(int i=0;i<bymod;i++){
41                 unsigned int ch=T[p].byte;
42                 unsigned char m=ch>>(8*(bymod-
i-1));
43                 fwrite(&m,1,1,fp_OS);
44             }
45             p=root;
46         }
47     }
48     buf=buf<<2;
49     pos=pos+2;
50 }

```



```

51     if(cnt==0)break;
52     fread(&buf,1,1,fp_IS);
53     pos=0;
54 }

```

同样根据四叉树的特性进行了改动。

• UI模块

```

1  int UI(){
2      int op;
3      cin>>op;
4      switch(op){
5          case 1:{
6              cout<<"选择压缩模式: "<<endl
7              <<"1.BIN_Compress"<<endl
8              <<"2.QUAD_Compress"<<endl;
9              short mode;
10             cin>>mode;
11             cout<<"选择压缩数据单位: "<<endl
12             <<"1.1byte/unit"<<endl
13             <<"2.2bytes/unit"<<endl;
14             short un;
15             cin>>un;
16             switch(mode){
17                 case 1:{
18                     char I_FILE_NAME[100];
19                     char O_FILE_NAME[100];
20                     cout<<"输入需要压缩的文件路径: "
21                     <<endl;
22                     cin>>I_FILE_NAME;
23                     cout<<"输入压缩后的文件输出路径: "
24                     <<endl;
25                     cin>>O_FILE_NAME;
26                     Compress_Bin(I_FILE_NAME,O_FILE_NAME,un);
27                     break;
28                 }
29                 case 2:{

```

```

28         char I_FILE_NAME[100];
29         char O_FILE_NAME[100];
30         cout<<"输入需要压缩的文件路径:  "
<<endl;
31         cin>>I_FILE_NAME;
32         cout<<"输入压缩后的文件输出路径:  "
<<endl;
33         cin>>O_FILE_NAME;
34
35         Compress_Quad(I_FILE_NAME,O_FILE_NAME,un);
36         break;
37     }
38     default:cout<<"error mode"<<endl;
39 }
40 return 1;
41 }
42 case 2:{
43     char I_FILE_NAME[100];
44     char name[100];
45     cout<<"输入需要解压缩的文件路径:  "<<endl;
46     cin>>I_FILE_NAME;
47     cout<<"输入解压缩后的文件名称:  "<<endl; //
48     无需后缀名
49     cin>>name;
50     Decompress(I_FILE_NAME,name);
51     return 1;
52 }
53 case 0:{
54     return 0;
55 }
56 default:{
57     cout<<"不合法的操作"<<endl;
58     return 1;
59 }
60 }

```


运行结果

压缩过程

```
输入要执行的操作：
1.压缩文件
2.解压文件
0.结束程序
1

输入要执行的操作：
1.压缩文件
2.解压文件
0.结束程序
1
选择压缩模式：
1.BIN_Compress
2.QUAD_Compress
2
选择压缩数据单位：
1.1byte/unit
2.2bytes/unit
2
输入需要压缩的文件路径：
test.zip
输入压缩后的文件输出路径：
zip.dat
total bytes: 10914941 Bytes
compression ratio: 99.9985%
```

目标文件内容

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded Text
00000000	04	34	2C	2E	7A	69	70	35	34	35	37	35	35	34	2C	36	. 4 , . z i p 5 4 5 7 5 5 4 , 6
00000010	35	35	33	36	2C	30	2C	32	2C	30	2C	35	34	38	2C	31	5 5 3 6 , 0 , 2 , 0 , 5 4 8 , 1
00000020	2C	39	32	2C	32	2C	38	39	2C	33	2C	31	31	37	2C	34	, 9 2 , 2 , 8 9 , 3 , 1 1 7 , 4
00000030	2C	31	30	38	2C	35	2C	31	34	36	2C	36	2C	31	30	32	, 1 0 8 , 5 , 1 4 6 , 6 , 1 0 2
00000040	2C	37	2C	31	33	37	2C	38	2C	31	31	35	2C	39	2C	31	, 7 , 1 3 7 , 8 , 1 1 5 , 9 , 1
00000050	34	35	2C	31	30	2C	31	30	39	2C	31	31	2C	31	32	32	4 5 , 1 0 , 1 0 9 , 1 1 , 1 2 2
00000060	2C	31	32	2C	39	39	2C	31	33	2C	31	30	32	2C	31	34	, 1 2 , 9 9 , 1 3 , 1 0 2 , 1 4
00000070	2C	31	30	32	2C	31	35	2C	38	34	2C	31	36	2C	38	32	, 1 0 2 , 1 5 , 8 4 , 1 6 , 8 2
00000080	2C	31	37	2C	31	31	35	2C	31	38	2C	38	38	2C	31	39	, 1 7 , 1 1 5 , 1 8 , 8 8 , 1 9
00000090	2C	38	32	2C	32	30	2C	39	31	2C	32	31	2C	37	38	2C	, 8 2 , 2 0 , 9 1 , 2 1 , 7 8 ,
000000A0	32	32	2C	31	31	35	2C	32	33	2C	37	35	2C	32	34	2C	2 2 , 1 1 5 , 2 3 , 7 5 , 2 4 ,
000000B0	39	38	2C	32	35	2C	31	30	30	2C	32	36	2C	31	30	37	9 8 , 2 5 , 1 0 0 , 2 6 , 1 0 7
000000C0	2C	32	37	2C	37	33	2C	32	38	2C	37	31	2C	32	39	2C	, 2 7 , 7 3 , 2 8 , 7 1 , 2 9 ,
000000D0	39	30	2C	33	30	2C	37	38	2C	33	31	2C	37	30	2C	33	9 0 , 3 0 , 7 8 , 3 1 , 7 0 , 3
000000E0	32	2C	38	39	2C	33	33	2C	38	31	2C	33	34	2C	39	32	2 , 8 9 , 3 3 , 8 1 , 3 4 , 9 2
000000F0	2C	33	35	2C	38	30	2C	33	36	2C	31	30	35	2C	33	37	, 3 5 , 8 0 , 3 6 , 1 0 5 , 3 7
00000100	2C	38	30	2C	33	38	2C	38	37	2C	33	39	2C	36	34	2C	, 8 0 , 3 8 , 8 7 , 3 9 , 6 4 ,
00000110	34	30	2C	38	39	2C	34	31	2C	31	30	35	2C	34	32	2C	4 0 , 8 9 , 4 1 , 1 0 5 , 4 2 ,
00000120	39	37	2C	34	33	2C	38	33	2C	34	34	2C	39	34	2C	34	9 7 , 4 3 , 8 3 , 4 4 , 9 4 , 4
00000130	35	2C	38	38	2C	34	36	2C	31	30	37	2C	34	37	2C	39	5 , 8 8 , 4 6 , 1 0 7 , 4 7 , 9
00000140	34	2C	34	38	2C	38	38	2C	34	39	2C	31	31	32	2C	35	4 , 4 8 , 8 8 , 4 9 , 1 1 2 , 5
00000150	30	2C	31	31	31	2C	35	31	2C	36	39	2C	35	32	2C	31	0 , 1 1 1 , 5 1 , 6 9 , 5 2 , 1
00000160	31	32	2C	35	33	2C	31	30	36	2C	35	34	2C	38	34	2C	1 2 , 5 3 , 1 0 6 , 5 4 , 8 4 ,
00000170	35	35	2C	37	30	2C	35	36	2C	38	38	2C	35	37	2C	39	5 5 , 7 0 , 5 6 , 8 8 , 5 7 , 9
00000180	38	2C	35	38	2C	37	33	2C	35	39	2C	39	35	2C	36	30	8 , 5 8 , 7 3 , 5 9 , 9 5 , 6 0
00000190	2C	36	39	2C	36	31	2C	38	36	2C	36	32	2C	37	32	2C	, 6 9 , 6 1 , 8 6 , 6 2 , 7 2 ,
000001A0	36	33	2C	37	38	2C	36	34	2C	39	32	2C	36	35	2C	31	6 3 , 7 8 , 6 4 , 9 2 , 6 5 , 1
000001B0	30	38	2C	36	36	2C	38	39	2C	36	37	2C	37	36	2C	36	0 8 , 6 6 , 8 9 , 6 7 , 7 6 , 6
000001C0	38	2C	38	33	2C	36	39	2C	37	32	2C	37	30	2C	37	39	8 , 8 3 , 6 9 , 7 2 , 7 0 , 7 9

解压缩过程

```
输入要执行的操作：
1.压缩文件
2.解压文件
0.结束程序
2
输入需要解压缩的文件路径：
zip.dat
输入解压缩后的文件名称：
result
---Decompression completed---
```

结果文件与源文件内容对比

3

5

