

# Huffman 编码压缩/解压器

## 一、问题描述

在合适的情况下，利用 Huffman 编码对文件进行压缩可以减少其占用空间，同时在需要使用到文件的时候也可以根据压缩文件中所提供的信息来将其还原为原文件。本次实验中，我们将实现一个基于 Huffman 编码的文件压缩/解压缩工具。

## 二、基本要求

基于 Huffman 编码实现一个压缩器和解压器（其中 Huffman 编码以字节作为统计和编码的基本符号单元），使其可以对任意的文件进行压缩和解压缩操作。针对编译生成的程序，要求压缩和解压缩部分可以分别独立运行。具体要求为：

- 每次运行程序时，用户可以指定**只压缩/只解压**指定路径的文件。实现的时候不限制与用户的交互方式，可供参考的方式包括但不限于
  - 根据命令行参数指定功能（压缩/解压缩）和输入/输出文件路径
  - GUI 界面
  - 运行程序后由用户交互输入指定功能和路径
- **【CAUTION!】不被允许的交互方式：**通过修改源代码指定功能和文件路径
- 压缩时不需要指定解压文件的目标路径，解压时不需要指定压缩前原文件的路径，压缩后的文件可以换到另一个位置再做解压缩

## 三、程序能实现的功能

1. 压缩指定路径的任意格式的文件
2. 解压指定路径的文件(无损)
3. 可以选择压缩的基本单元(0.5Byte ~ 4Byte)
4. 可以选择使用多元 Huffman 数进行压缩 (2~16)
5. 3, 4 项同时使用

## 四、实验思路

### 1. 压缩

- 第一遍扫描文件，根据用户选择的压缩基本单元，统计每一种基本单元出现的频次
- 以出现的频次作为权重，根据用户选择构建 Huffman 树，若用户选择多元压缩，则首先补全节点数，其次使用优先队列，每次弹出最小的元素，构造 Huffman 树
- 根据构造出的 Huffman 树，求得 Huffman 编码 (压缩单元较小时，使用指针 new 内存的形式加速，压缩单元较大时，使用 map 来控制内存的使用)
- 首先输出被压缩文件的类型，压缩时使用的基本单元大小，Huffman 树种类，再输出每一个基本单元出现的频次
- 再次扫描文件，每当获取一个基本单元大小的字节，二进制输出它的 Huffman 编码 (每八位以一个字符的形式输出)

### 2. 解压

- 首先获得压缩文件头部的相关信息，以及各个基本单元出现的频次
- 用和压缩时完全相同的方式构造出 Huffman 树，获得 Huffman 编码
- 扫描剩下的文件内容，根据二进制编码，转化为原来的基本单元输出

## 五、项目结构

```
.
|-- CMakeLists.txt
|-- include
|   |-- big.h
|   |-- huffman.h
|   |-- small.h
|   `-- tempwindow.h
`-- src
    |-- big.cpp
    |-- huffman.cpp
    |-- main.cpp
    |-- small.cpp
    `-- tempwindow.cpp
```

## 六、不同文件的具体内容

### 1. main.cpp

- 功能界面，实现与用户的交互

### 2. huffman.h和huffman.cpp

实现构造 Huffman 树和 Huffman 编码

- 存储 Huffman 树的节点

```
1  typedef struct HTNode
2  {
3      long long int num;           //表示第几个元素
4      long long int key;          //对应元素的值
5      int weight;                 //权值
6      long long int parent;       //双亲
7      long long int child[16];    //孩子
8      friend bool operator<(HTNode a, HTNode b)
9      {
10         return a.weight > b.weight;
11     }
12 } HTNode, *HuffmanTree;
```

- 各函数的定义

```
1  void init(HuffmanTree &HT, std::map<long long int, int> &map, char
   **&HC, int tree_n);
2  void init_big(HuffmanTree &HT, std::map<long long int, int> &map,
   std::unordered_map<long long int, char *> &HC, int tree_n);
3  void init_for_de(HuffmanTree &HT, std::map<long long int, int> &map,
   int tree_n);
4  void output_huffmantree(HuffmanTree &HT, std::map<long long int, int>
   &map, int tree_n);
5  int judge(int a);
```

■ void init(HuffmanTree &HT, std::map<long long int, int> &map, char \*\*&HC, int tree\_n)

实现压缩基本单元较小时 Huffman 树的生成

```
1 void init(HuffmanTree &HT, std::map<long long int, int> &map,
2 char **&HC, int tree_n)
3 {
4     std::cout << "正在构造Huffman树: " << std::endl;
5     //初始化 huffman树
6     long long int fre = map.size();
7     HT = new HTNode[2 * fre];
8     for (long long int i = 0; i < 2 * fre; i++)
9     {
10         HT[i].num = i;
11         HT[i].key = 0;
12         HT[i].weight = 0;
13         for (int j = 0; j < tree_n; j++)
14             HT[i].child[j] = 0;
15         HT[i].parent = 0;
16     }
17     int index = 1;
18     HT[0].weight = INT_MAX;
19     for (auto iter : map)
20     {
21         HT[index].key = iter.first;
22         HT[index].weight = iter.second;
23         index++;
24     }
25     //使用优先队列构造huffman树
26     std::priority_queue<HTNode> ans;
27     for (int i = 1; i <= fre; i++)
28         ans.push(HT[i]);
29     if (tree_n != 2)
30     {
31         int blank;
32         if (fre % (tree_n - 1) == 0)
33             blank = 1;
34         else
35             blank = tree_n - fre % (tree_n - 1);
36         if (blank != tree_n - 1)
37         {
38             HuffmanTree Tree_blank = new HTNode;
39             Tree_blank->key = 0;
40             Tree_blank->parent = 0;
41             Tree_blank->weight = 0;
42             for (int j = 0; j < tree_n; j++)
43                 Tree_blank->child[j] = 0;
44             for (int i = 0; i < blank; i++)
45             {
46                 Tree_blank->num = fre + i + 1;
47                 ans.push(*Tree_blank);
48             }
49         }
50     }
51     //要从加了0之后开始计算 即修正fre
52     for (long long int now = fre + 1; now < 2 * fre; now++)
```

```

53     {
54         int now_weight = 0;
55         for (int i = 0; i < tree_n; i++)
56         {
57             HTNode temp_Node = ans.top();
58             HT[temp_Node.num].parent = now;
59             HT[now].child[i] = temp_Node.num;
60             now_weight += temp_Node.weight;
61             ans.pop();
62         }
63         HT[now].weight = now_weight;
64         if (ans.empty())
65             break;
66         ans.push(HT[now]);
67     }
68     //构造 huffman编码
69     char *cd = new char[fre];
70     cd[fre - 1] = '\0';
71     for (long long int i = 1; i <= fre; i++)
72     {
73         long long int start = fre - 1;
74         long long int j, p;
75         for (j = i, p = HT[i].parent; p != 0; j = p, p =
HT[p].parent)
76         {
77             int index = 0;
78             while (HT[p].child[index] != j)
79                 index++;
80             int wei = judge(tree_n);
81             for (int i = 0; i <= wei - 1; i++)
82                 cd[--start] = ((index >> i) & 1) + '0';
83         }
84         HC[HT[i].key] = (char *)malloc((fre - start) *
sizeof(char));
85         strcpy(HC[HT[i].key], &cd[start]);
86     }
87 }

```

- **void init\_big(HuffmanTree &HT, std::map<long long int, int> &map, std::unordered\_map<long long int, char \*> &HC, int tree\_n)**

实现压缩基本单元较小时 Huffman 树的生成，代码实现与较小时基本相同

- **void init\_for\_de(HuffmanTree &HT, std::map<long long int, int> &map, int tree\_n)**

解压时用于构造 Huffman 树，由于只需要构造 Huffman 树，而不需要 Huffman 编码 (可以根据树一步步查找)，所以删去求 Huffman 编码的过程，其余内容一致

- **void output\_huffmantree(HuffmanTree &HT, std::map<long long int, int> &map, int tree\_n)**

展示 Huffman 树

```

1
2 void output_huffmantree(HuffmanTree &HT, std::map<long long
int, int> &map, int tree_n)
3 {
4     if (map.size() == 1)

```

```

5         std::cout << "只有一个元素!" << std::endl;
6     else
7     {
8         std::cout << "Huffman树:" << std::endl;
9         printf("      | key | weight | parent | ch\n");
10        long long int fre = map.size();
11        for (long long int i = 1; i < 2 * fre; i++)
12        {
13            printf("%6lld | %6lld | %6d | %6lld ", i,
14                HT[i].key, HT[i].weight, HT[i].parent);
15            for (int j = 0; j < tree_n; j++)
16                printf("| %6lld ", HT[i].child[j]);
17            putchar('\n');
18            if (HT[i].parent == 0)
19                break;
20        }
21        getchar();
22    }

```

#### ■ int judge(int a)

根据 Huffman 树节点孩子的个数，判断需要使用多少位的编码

```

1  int judge(int a) //判断编码
2  {
3      if (a <= 2)
4          return 1;
5      else if (a > 2 && a <= 4)
6          return 2;
7      else if (a > 4 && a <= 8)
8          return 3;
9      else
10         return 4;
11 }

```

### 3. small.h和small.cpp

实现压缩过程

- 类的定义

```

1  class small
2  {
3  public:
4      small();
5      ~small();
6      void compress();
7      bool compress_input();
8      void compress_output();
9      void compress_output_big();
10
11 private:
12     int select;           //压缩单位选择
13     int tree_n;           //n元Huffman树
14     float size;           //文件大小
15     int char_size;       //防止读取出现错误

```

```

16     int now_byte;                                //当前读取到的的文
    件位数
17     std::string data_path;                        //需要压缩的文件的
    路径
18     std::string ans_path;                        //需要存储的完成压
    缩的路径
19     std::string type;                            //用来存储被压缩的
    文件的格式
20     std::map<long long int, int> map;            //需要压缩的文件中
    不同字符的数量
21     std::unordered_map<long long int, char *> HC; //不同字符对应的
    huffman编码
22     char **HC;                                    //小数据量用数组
23     HuffmanTree HT;                              //huffman树
24 };

```

#### ■ void compress()

与用户交互，压缩相关选项的选择

#### ■ bool small::compress\_input()

统计基本单元出现的次数，具体计数部分如下所示 (需要注意的是最后可能有尚未满足一个基本单元的内容，未满足的部分用 0 替代 / 需要特判只有一个基本单元的情况)

```

1  char c;
2  int every = 8 * select * 0.5; //单位 bit
3  long long int tt_key = 0;      //用来记录 key
4  int now_bit = 0;              //用来记录当前读入了多少个bit
5  int char_now = 0;             //用来记录当前读取的字符使用到的 bit
6  std::unordered_map<long long int, int> temp_map;
7  while (infile.get(c))
8  {
9      char_size++;
10     char_now = 0;
11     while (char_now < 8)
12     {
13         while (char_now < 8)
14         {
15             tt_key += (long long int)(((int)c >> (7 -
char_now)) & 1) << (every - 1 - now_bit);
16             char_now++;
17             now_bit++;
18             if (now_bit == every)
19                 break;
20         }
21         if (now_bit == every)
22         {
23             if (temp_map.count(tt_key) == 0)
24             {
25                 map[tt_key] = 1;
26                 temp_map[tt_key] = 1;
27             }
28             else
29                 map[tt_key]++;
30             now_bit = 0;
31             tt_key = 0;
32         }

```

```

33     }
34 }
35 if (now_bit != 0)
36 {
37     if (temp_map.count(tt_key) == 0)
38         map[tt_key] = 1;
39     else
40         map[tt_key]++;
41 }

```

#### ■ void small::compress\_output()

先输出相关基本信息，接着根据 Huffman 编码，对文件进行压缩，具体压缩部分如下

```

1  char c;
2  int every = 8 * select * 0.5; //单位 bit
3  int tt_key = 0;                //用来记录 key
4  int now_bit = 0;                //用来记录当前读入了多少个bit
5  int char_now = 0;              //用来记录当前读取的字符使用到的 bit
6  while (infile.get(c))
7  {
8      now_byte++;
9      char_now = 0;
10     while (char_now < 8)
11     {
12         while (char_now < 8)
13         {
14             tt_key += (((int)c >> (7 - char_now)) & 1) <<
(every - 1 - now_bit);
15             char_now++;
16             now_bit++;
17             if (now_bit == every)
18                 break;
19         }
20         if (now_bit == every)
21         {
22             i = 0;
23             while (HC[tt_key][i] != '\0')
24             {
25                 tt += (HC[tt_key][i] - '0') << (7 - num);
26                 i++;
27                 num++;
28                 if (num == 8)
29                 {
30                     outfile.put((char)tt);
31                     num = 0;
32                     tt = 0;
33                 }
34             }
35             tmp_struct = {now_byte, size}; //实时更新当前的进度
36             now_bit = 0;
37             tt_key = 0;
38         }
39     }
40 }
41 //输出最后一个可能没有到达一个单位的字符
42 if (now_bit != every && now_bit != 0)

```

```

43 {
44     i = 0;
45     while (HC[tt_key][i] != '\0')
46     {
47         tt += (HC[tt_key][i] - '0') << (7 - num);
48         i++;
49         num++;
50         if (num == 8)
51         {
52             outfile.put((char)tt);
53             num = 0;
54             tt = 0;
55         }
56     }
57 }
58 //输出可能没有到达8bit的最后一个字符
59 if (num != 0)
60     outfile.put((char)tt);
61 tmp_struct = {now_byte, size};

```

#### ■ void small::compress\_output\_big()

所选基本单元较大时使用函数，具体实现基本相同

### 4. big.h和big.cpp

实现解压过程

- 类的定义

```

1 class big
2 {
3 public:
4     big();
5     void decompress();
6     bool decompress_input_output();
7
8 private:
9     int select; //压缩单位选择
10    int tree_n; //n元Huffman树
11    float size; //文件大小
12    int now_byte; //已经读取的字符数
13    std::string data_path; //需要解压的文件的路径
14    std::string ans_path; //解压完成的文件路径
15    std::string type; //用来存储被压缩的文件的格式
16    std::map<long long int, int> map; //需要解压的文件中不同字符的数量
17    HuffmanTree HT; //huffman树
18 };

```

#### ■ void decompress()

与用户进行交互

#### ■ bool decompress\_input\_output()

解压时，输入与输出在一个函数实现 (特判只有一个基本单元的情况) 解压关键部分如下所示

```

1 char ans[8];
2 char c;

```



```

3 int index = 0;
4 int fre = map.size();
5 int root_loc;
6 int now;
7 int now_bit = 0; //记录当前输出字符到的bit
8 int out_tmp = 0; //用来记录输出的字符
9 int wei = judge(tree_n);
10 int wei_now = 0;
11 int switch_child = 0;
12 for (root_loc = 1; root_loc < 2 * fre; root_loc++)
13 {
14     if (HT[root_loc].parent == 0)
15         break;
16 }
17 now = root_loc;
18 while (1)
19 {
20     infile.get(c);
21     int tt = c;
22     index = 0;
23     for (int i = 7; i >= 0; i--)
24         ans[index++] = ((tt >> i) & 1) + '0';
25     index = 0;
26     while (1)
27     {
28         switch_child += (ans[index] - '0') << (wei - 1 -
29         wei_now);
30         wei_now++;
31         if (wei == wei_now)
32         {
33             wei_now = 0;
34             now = HT[now].child[switch_child];
35             switch_child = 0;
36         }
37         index
38         if (HT[now].child[0] == 0)
39         {
40             //获取对应权值对应的 bit位
41             int int_to_char = HT[now].key;
42             char ans_[every + 1];
43             ans_[every] = '\0';
44             for (int j = 0; j < every; j++)
45                 ans_[j] = ((int_to_char >> (every - 1 - j)) &
46                 1) + '0';
47             int j = 0;
48             while (ans_[j] != '\0')
49             {
50                 now_bit++;
51                 out_tmp += (ans_[j] - '0') << (8 - now_bit);
52                 if (now_bit == 8)
53                 {
54                     char_size--;
55                     outfile.put(out_tmp);
56                     now_byte++;
57                     tmp_struct = {now_byte, size};
58                     out_tmp = 0;
59                     now_bit = 0;
60                 }
61             }
62         }
63     }
64 }

```

```

59         j++;
60         if (char_size == 0)
61             break;
62     }
63     now = root_loc;
64 }
65 if (index == 8 || char_size == 0)
66     break;
67 }
68 if (char_size == 0)
69     break;
70 }

```

## 5. tempwindow.h和tempwindow.cpp

多线程，命令行绘制进度界面

- 用于传值的结构体

```

1 struct poss_help
2 {
3     int now_b;
4     float total;
5 };

```

- 具体绘制函数

```

1 void *possession(void *threadarg)
2 {
3     while (1)
4     {
5         struct poss_help *p = (struct poss_help *)threadarg;
6         int now_byte = p->now_b;
7         float size = p->total;
8         int temp = 40 * (now_byte / size);
9         int temp_n = 40 - temp;
10        putchar('\r');
11        putchar '[';
12        for (int i = 0; i < temp; i++)
13            putchar('#');
14        for (int i = 0; i < temp_n; i++)
15            putchar(' ');
16        std::cout << "]" << (int)(100 * (now_byte / size)) << "%
17";
18        if ((100 * (now_byte / size)) > 99)
19        {
20            putchar('\r');
21            putchar '[';
22            for (int i = 0; i < 40; i++)
23                putchar('#');
24            std::cout << "]" << 100%;
25            break;
26        }
27        //因为字节可能计算失误 可能会导致进度条最后跳一下
28    }
29    pthread_exit(NULL);
30 }

```

- 多线程的调用

- 建立

```
1 pthread_t threads;  
2 poss_help tmp_struct;  
3 tmp_struct = {now_byte, size};  
4 pthread_create(&threads, NULL, possession, (void *)&tmp_struct);
```

- 结束

```
1 pthread_join(threads, NULL);
```

## 七、编译过程

- 编译环境
  - Ubuntu-20.04 ( wsl2 )
  - gcc 9.3.0
  - cmake version 3.16.3
- CMakeLists.txt (注意修改Debug模式和Release模式)




```
1 cmake_minimum_required(VERSION 3.5)  
2  
3 project(yasuo)  
4  
5 set (CMAKE_CXX_STANDARD 17)  
6  
7 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread" )  
8  
9 set(SOURCES  
10     src/main.cpp  
11     src/big.cpp  
12     src/small.cpp  
13     src/tempwindow.cpp  
14     src/huffman.cpp)  
15  
16 add_definitions(-w)# system忽略了返回值, 消除编译警告  
17  
18 add_executable(yasuo ${SOURCES})  
19  
20 SET(CMAKE_BUILD_TYPE "Release")  
21  
22 target_include_directories(yasuo  
23 PRIVATE  
24     ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

- 编译操作(Release版本为例, 首先进入项目目录)

```
1 mkdir Release  
2 cd Release  
3 cmake ..  
4 make  
5 ./yasuo  
6 # 运行程序
```

## 八、运行效果




- 使用1.5Byte为基本单元，二叉Huffman树压缩png文件，源文件258kb，压缩后182kb，解压后258kb，效果较为明显，同时文件解压后无任何变化

 lena.dat	2021/12/21 23:57	DAT 文件	182 KB
 lena.png	2021/11/16 17:09	PNG 文件	258 KB
 lena_.png	2021/12/21 23:57	PNG 文件	258 KB

- 性能测试

使用该程序压缩大小1.3G的txt文档

- 压缩用时：130.917s
- 解压用时：113.172s
- 压缩效率：46%

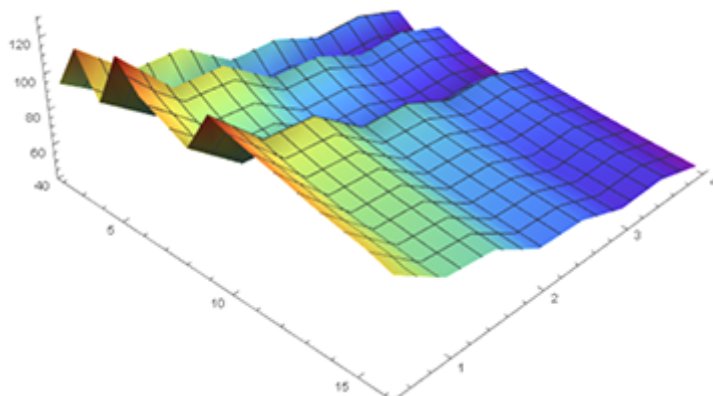
 distance_info.dat	2021/12/22 9:43	DAT 文件	598,056 KB
 distance_info.txt	2021/12/14 23:27	文本文档	1,297,600...
 distance_info_.txt	2021/12/22 10:09	文本文档	1,297,600...

## 九、关于Huffman压缩的探索

- 压缩一包含中英文及数字的txt文档，压缩效果如下(压缩率= $\frac{\text{压缩文件大小}}{\text{原始文件}}$ )

压缩率 \ 基本单元	0.5Byte	1.0Byte	1.5Byte	2.0Byte	2.5Byte	3.0Byte	3.5Byte	4.0Byte
n 元树								
2	94.8%	75.0%	74.3%	58.3%	57.6%	48.2%	49.0%	41.7%
3	120.6%	95.1%	94.2%	73.9%	72.9%	61.0%	61.6%	52.5%
4	98.0%	75.9%	75.0%	58.8%	58.0%	48.6%	49.3%	42.0%
5	129.4%	98.7%	97.0%	76.5%	75.1%	62.9%	63.4%	54.1%
6	115.0%	90.3%	87.8%	68.9%	67.8%	56.7%	57.3%	48.8%
7	108.6%	83.4%	81.2%	63.6%	62.5%	52.4%	52.0%	45.1%
8	101.7%	78.0%	76.1%	59.7%	58.7%	49.1%	49.8%	42.4%
9	126.1%	98.6%	96.3%	75.4%	74.3%	62.1%	62.5%	53.3%
10	122.0%	94.0%	92.0%	72.2%	71.0%	59.5%	59.8%	51.1%
11	118.1%	90.2%	88.6%	69.5%	68.2%	57.3%	57.6%	49.3%
12	114.7%	87.3%	85.8%	67.3%	66.0%	55.4%	55.8%	47.6%
13	110.7%	85.0%	83.4%	65.5%	64.1%	52.7%	54.3%	46.2%
14	107.6%	83.7%	81.2%	64.0%	62.5%	52.3%	52.9%	44.9%
15	104.8%	81.9%	79.3%	62.5%	61.1%	51.0%	51.6%	43.8%
16	100.0%	80.8%	77.6%	61.0%	59.7%	49.8%	50.5%	42.9%

- 利用上述数据作图 (mathematica作图有一定偏差)



- 根据上述实验数据可以看出:
  - 压缩基本单元为0.5Byte时, 基本没有压缩效果
  - 随着压缩基本单元的增大, 压缩效果总体上呈现上升趋势
  - 在压缩的基本单元从3.0Byte变为3.5Byte时, 压缩效果均有小幅下降
  - 当选用 $2^n$ 叉Huffman树时, 压缩效果明显更佳, 而当选用 $2^n + 1$ 叉Huffman树时, 压缩效果较差 ( $2^n$ 叉Huffman树每一个节点都刚好可以利用  $n$  bit的Huffman编码)

**因此, 在压缩txt文本文件时, 选用 $2^n$ 叉Huffman树, 在综合时间的情况下选择更大的压缩基本单元, 能够提高压缩效率**

同时值得指出的是, 对于图片, 视频等文件类型, 基本单元极有可能发生出现次数相近的情况, 此时综合时间等考虑, 可以尽量选择1Byte为压缩的基本单元

## 十、实验收获

- 更加熟悉了Huffman树, 及相关算法
- 对Huffman压缩时选用的基本单元及 $n$ 叉Huffman树有了全新的认识
- 提升了文件操作, 位运算的能力