

数据结构：哈夫曼编码压缩、解压文件 实验报告

毛子恒 李臻 张梓靖

2020 年 11 月 23 日

小组成员

班级：2019211309	姓名：毛子恒	学号：2019211397	分工：代码 文档
班级：2019211310	姓名：李臻	学号：2019211458	分工：测试 文档
班级：2019211308	姓名：张梓靖	学号：2019211379	分工：可视化 文档

目录

1 需求分析	2
2 概要设计	6
3 详细设计	6
4 调试分析报告	9
5 用户使用说明	10
6 测试结果	10
7 可视化	13

1 需求分析

1.1 题目描述

对于一个指定的文件，应用哈夫曼编码压缩文件，或者对本程序生成的压缩文件进行解压。

1.2 输入描述

程序从给定的二进制文件中读入数据。

1.3 输出描述

程序向给定的二进制文件中输出结果，向标准输出中输出提示。

压缩文件分为若干个部分，每个部分包含如下信息：

1. 原字符串中包含的不同字符的个数；
2. 哈夫曼树中每个节点存储的字符（非叶结点不存储字符）、结点左右孩子的序号；
3. 原字符串的长度、压缩后位串的长度；
4. 编码后的位串。

输出分为五种情况：

1. 输入合法，程序正常运行结束，此时结果存储在给定的二进制文件中，并在标准输出中打印提示信息。
2. 输入不合法，程序向标准输出打印错误信息，并且要求重新输入或异常退出。
3. 程序发生运行时错误，比如内存分配失败。此时程序没有输出。

关于更多细节请参考 5 用户使用说明部分。

1.4 样例输入输出

由于压缩后的二进制文件难以阅读，故通过转换程序（./bit2string.c）将该压缩文件转化成容易阅读的形式展现在本实验报告中，将不同数据以空格或换行符分隔，原本按每 8 位存储的位串转化成 01 字符串。

1.4.1 样例输入输出 1

压缩。

【输入】

```
aaababcd
```

【输出】

（'#' 后的内容为注释）

```
4 # 出现的不同字符的个数/哈夫曼树叶结点的个数
c # 一个叶结点，其中存储字符c，没有左右孩子
d
b
```

```

a
1 2 # 一个非叶结点，不存储字符，左右孩子分别是1、2
3 5
4 6
8 2 # 原串的长度，压缩后字符串的长度（2个字节）
00010010110111100 # 压缩后的位串

```

1.4.2 样例输入输出 2

压缩。

【输入】

```
!#$"#$"$!"!$$$#" "$!$$!"!$!#"!"#"!!#"##!$$$#" "$"$#$!$$"#!"#$##!!"##!##!#$!" "$!"!#$
```

【输出】

```

4
#
!
"
$
1 2
3 4
5 6
100 25
0100111010110011110111110010110011111001011001011101101110011011001110100100001101000100101010010000010011111111000
101110111011000111110110001100011000001011000100000100111001101011011001001111

```

1.4.3 样例输入输出 3

压缩。

【输入】

```
1>8#2$'&'!-).1%2/1:)/,8;2!3>(/69",#&7;>#$</'!-=>+2#5&>!)5;(<96&;>0,7,;:/.)7$$/%5-(67933<(';==4464!12
```

【输出】

```

29
"
+
0
.
:
8
-
5
9
<
!
4

```

```
3
%
)
,
&
$
#
=
7
6
1
,
(
2
>
;
/
1 2
3 30
4 5
6 31
7 8
9 10
11 12
13 14
15 32
16 17
18 19
20 21
22 23
24 25
26 33
27 34
28 35
29 36
37 38
39 40
41 42
43 44
45 46
47 48
49 50
51 52
53 54
55 56
100 60
111110100001101101100101101011000110011100001010101101011101111110101001010001111110111110110100000000011001100010
100101010001000001100011110011100011110000011011110011110101100100110111101001111100010101110001110001010111000100
001111100101101101011110010100100101011001011011000010111101110111101100101100100001110000011101000001101011111000
```

```
10111010110111011101011010100010101010101000011110111010111010100101000111100011100001101110011100100111001111
110100111001011111001000
```

1.4.4 样例输入输出 4

解压。

【输入】

```
4
c
d
b
a
1 2
3 5
4 6
8 2
0001001011011100
```

【输出】

```
aaababcd
```

1.4.5 样例输入输出 5

解压。

【输入】

```
4
#
!
"
$
1 2
3 4
5 6
100 25
0100111010110011101111100101100111110010110010111011011100110110011010010000110100010010101001000001001111111000
10111011101100011111011000110001100000101100010000100111001101011011001001111
```

【输出】

```
!#$"#$"$!"!$$$#$"$!$#$!"!$!#">#!"#!!#$$$#"#$"$#$!$#$!"#$$$!"#"$!"!#$
```

1.5 程序功能

对于压缩过程，程序构造哈夫曼树并且对文件进行压缩，将需要的解码信息和压缩后的位串保存到压缩文件中。
对于解压过程，程序从压缩文件中读入解码信息和位串，并利用哈夫曼树进行解压。

2 概要设计

2.1 问题解决的思路

构造哈夫曼树进行压缩、解压过程。

2.2 哈夫曼编码的设计

2.3 主程序的流程

1. 获取输入，判断是否合法
2. 如果合法，则调用压缩/解压函数

2.4 各程序模块之间的层次关系

程序模块层次关系图如图 1。

图 1: 程序模块层次关系

3 详细设计

3.1 哈夫曼编码的实现

哈夫曼编码的设计中基本操作的伪代码算法如下：

3.2 函数的调用关系图

函数调用关系图如图 2。

图 2: 函数调用关系图

4 调试分析报告

4.1 调试过程中遇到的问题和思考

程序在一些小细节中出现了问题，比如 `val` 数组需要多分配一个空间，矩阵相乘后 `c` 矩阵的列数等于 `b` 矩阵的列数。

初次实现矩阵相加算法时设计有误，使得两个矩阵的某个对应位置只要有一个为零，就会忽略这个位置，这个问题通过 `if` 判断修复。

在随机测试中发现数据规模较大时结果很容易超出 `int` 类型的范围，故矩阵元素采用 `long long` 类型。

4.2 设计实现的回顾讨论

由于二维数组的内存分配、函数传参较为复杂，所以输入、输出时使用一维数组存储的矩阵。期望矩阵的规模不超过 10^3 ，所以 `MATRIXINCREASESIZE` 常量的值设置为 10^5 。

4.3 算法复杂度分析

`initMatrix`, `expandMatrix`, `destroyMatrix` 函数的复杂度为 $O(1)$

`array2Matrix`, `matrix2Array`, `addMatrix` 函数的复杂度为 $O(n^2)$ ，`mulMatrix` 函数的复杂度为 $O(n^3)$ ，视矩阵的稀疏程度，算法的时间复杂度会有常数级别的优化。在本报告的最后部分有讨论。

主函数的时间复杂度为 $O(n^2)$ ，整体时间复杂度为 $O(n^3)$ 。

整体空间复杂度为 $O(n^2)$ 。

4.4 改进设想的经验和体会

4.4.1 改进 1

可以在输入/输出时简化数组存储的矩阵和三元组表存储的矩阵之间转化的过程，会有常数级别的优化。

5 用户使用说明

使用 `gcc` 编译生成可执行文件。

```
gcc -o main -std=c11 main.c matrix.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后通过标准输入输入数据，输入格式参考 1.2 节的输入描述，结果通过标准输出返回。如果输入合法并且程序正常运行结束，主函数返回值为 0。

6 测试结果

测试环节分为三个步骤。

6.1 测试第一部分

对 1.4 节给出的样例进行测试。

6.2 测试第二部分

测试边界条件。

【输入】

```
1 0
1 1
2
```

【输出】

Please check your input.

【输入】

```
4 3
7 0 6
0 0 5
1 0 0
0 0 2
3 3
5 7 0
0 3 0
1 3 5
```

【输出】

Cannot add matrix A and B, An != Bn.
41 67 30
5 15 25
5 7 0
2 6 10

【输入】

```
3 3
7 0 6
1 0 4
1 0 0
3 4
5 7 0 1
0 3 0 0
1 3 5 0
```

【输出】

Cannot add matrix A and B, Am != Bm.
41 67 30 7
9 19 20 1
5 7 0 1

【输入】

```
3 4
7 0 6 4
1 0 4 0
1 0 0 6
3 4
```



```
5 7 0 1
0 3 0 0
1 3 5 0
```

【输出】

```
12 7 6 5
1 3 4 0
2 3 5 6
Cannot multiply matrix A and B, Am != Bn.
```

6.3 测试第三部分

将原解法与二维数组实现的矩阵加法和乘法 (testing/test.c) 比对。

测试在 macOS Catalina 10.15.6 下进行。

在 $n \leq 10$, $n \leq 100$, $n \leq 1000$ 的范围下分别随机生成 1000 组测试数据, 分别传入 main 和 test, 并且比对两程序的输出。

3000 组数据中两程序的输出均相同。

数据生成程序 (testing/data.cpp) 如下:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int SIZE = 1e3, PERCENT = 50;
5
6 int main()
7 {
8     srand(time(0));
9     int n = rand() % SIZE + 1;
10    printf("%d %d\n", n, n);
11    for (int i = 1; i <= n; ++i)
12    {
13        for (int j = 1; j <= n; ++j)
14            printf("%d ", rand() % 100 > PERCENT ? 0 : rand() % 1000 + 1);
15        puts("");
16    }
17    printf("%d %d\n", n, n);
18    for (int i = 1; i <= n; ++i)
19    {
20        for (int j = 1; j <= n; ++j)
21            printf("%d ", rand() % 100 > PERCENT ? 0 : rand() % 1000 + 1);
22        puts("");
23    }
24    return 0;
25 }
```

比对脚本 (testing/chk.sh) 如下:

```
1 for i in {1..1000}
2 do
```

```
3     sleep 1
4     ./data >in.in
5     ./main <in.in >out.out
6     ./test <in.in >out1.out
7     if ! diff out.out out1.out
8     then
9         break
10    fi
11    echo "Correct"
12 done
```

7 可视化

随机生成若干组数据，由三元组表实现和二维数组实现分别计算，比对运行时间，并且使用 JavaScript 将结果可视化。

比对脚本 (testing/timecount.py) 如下：

```
1 import os
2 import json
3 import time
4 a = []
5 b = []
6 for i in range(100):
7     os.system("./data >in.in")
8     starttime = time.time()
9     os.system("./main <in.in >out.out")
10    endtime = time.time()
11    a.append(endtime-starttime)
12    starttime = time.time()
13    os.system("./test <in.in >out1.out")
14    endtime = time.time()
15    b.append(endtime-starttime)
16    print(i)
17
18 json_str = json.dumps([a, b])
19 with open("80%result.json", mode="w") as file:
20     file.write(json_str)
```

数据规模如下：

$n = 1000$ ，20% 的元素是 0、50% 的元素是 0、80% 的元素是 0 的数据各 100 组，共 300 组数据。

7.0.1 实现细节

利用已有的时间数据，使用 Highcharts 库绘制二维柱状图，比较两种算法的运行时间差距。

7.0.2 用户使用说明

使用现代浏览器打开 Chart/index.html，即可看到柱状图，将鼠标指针放到某个柱上可以看到对应的值。

7.0.3 示例

见图 3。

图 3: 可视化示例

7.0.4 结论

可以看出在规模较大的数据中，相比二维数组的实现，三元组表的实现在绝大多数情况下都有明显的性能优势，并且当矩阵越稀疏时性能优势越明显。