实验三——Huffman 压缩软件

PB17000002

古宜民

2018.11.27

1. 实验基本要求

编写基于 Huffman 编码的压缩软件，实现文件（文本、二进制文件）的压缩和解压缩功能，并实现图形界面。

2. 实验实现内容

完整代码见：

https://github.com/ustcpetergu/Datastructure/tree/master/Course2018Spring/Huffman

使用 C++实现了输入文件计算 Huffman 编码、根据 Huffman 编码压缩文件、解压缩文件的功能，使用 Python 的 Tkinter 库绘制简易的图形界面。

3. 实验代码结构和关键代码讲述

数据结构：定义了 HuffmanTree 和 HuffmanCode 两个类。HuffmanTree 用于存放 Huffman 树，使用长度为 2n-1 的数组存放，其中只有一个主要方法，即在得到了各个叶子节点权值的条件下建立 Huffman 树。HuffmanCode 用于存放和处理 Huffman 编码。Huffman 编码用一维指针数组存放，每个元素为 bool*，指向动态分配的存储 Huffman 编码中 0 和 1 的一块空间。每块空间的长度在另一个数组 len 中存放。类只有一个主要方法，即输入一个构建好的 Huffman 树，计算出 Huffman 编码并存储。本程序中，以每个字节（8bits）为单位计算出现频率和 Huffman 编码。

压缩过程：

先将文件读一次，计算出每个字符出现的次数作为权值存储在 Huffman 树的叶子节点内。同时统计文件总字节数。

关键代码：`tree.tree[(unsigned char)chr].weight++;`

写文件时，先写入标识字符串和 Huffman 树以及文件总字节数，然后一边一个字节一个字节读入文件，一边计算 Huffman 编码并写入压缩文件。编写代码过程中发现，加设写缓存能明显加快写速度。默认使用 1M 写缓存。写满写缓存后一次性写入 1M 字符。文件结束后，如果还有剩余的几个字节或是未能凑成整个字节没有写入文件，统一补 0 并写入。

解压缩过程：

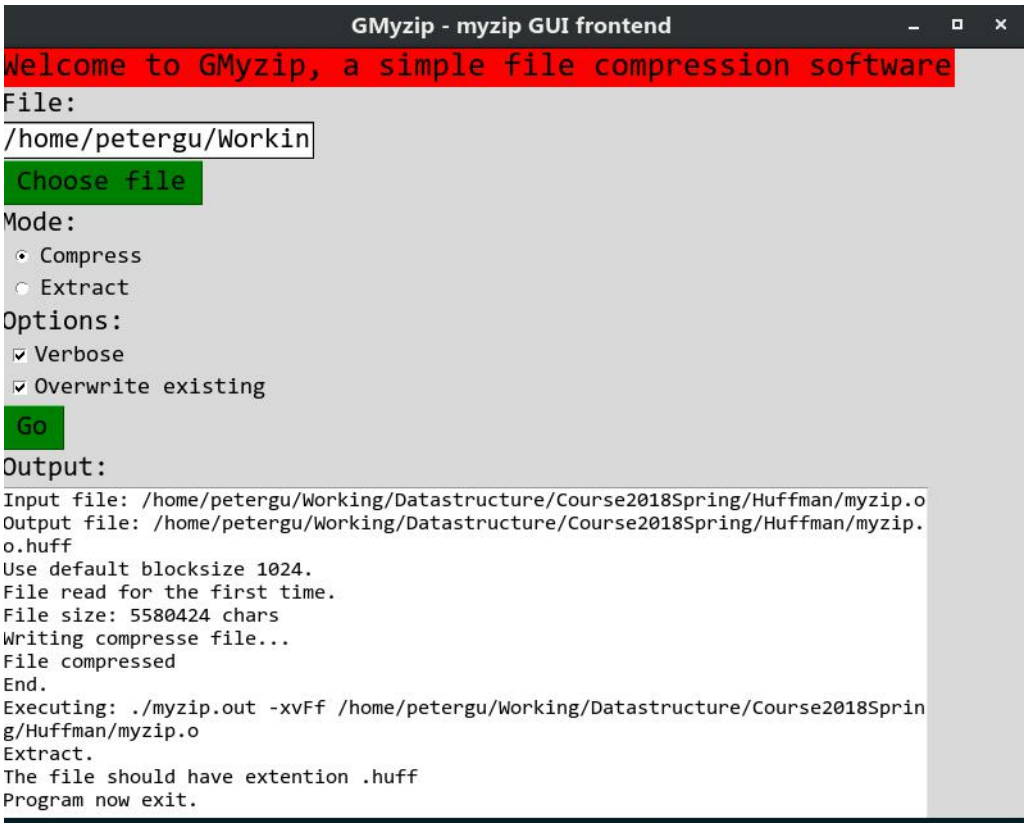
先读入文件标识信息和 Huffman 树。之后同时读入字节并遍历 Huffman 树求 Huffman 编码对应的字符并存入缓存，写满缓存后统一写入文件。因为之前存入了文件总字节数，所以解码了足够的字节数后就完成解压缩过程，多出来的几个 bit 为之前补的 0，忽略即可。

速度和效果：压缩可达 5M/s，解压可达 14M/s。编译时使用-O3，则可达压缩 23M/s，解压 18M/s。压缩后，文本、视频、音频压缩率约为 81%，99%，92%，而图片和 zip 压缩包都略有增大，相比专业压缩软件 gzip 等有一定差距，尤其是文本。

接口和图形界面：C++程序提供命令行参数接口，Tk 前端仅仅是对其进行了包装。命令行参数解析使用了 jarro2783/cxxopts 项目。参数列表：

```
("v,verbose", "Being verbose")
("c,compress", "Compress file")
("x,extract", "Extract compressed file")
("F,force", "Overwrite existing file")
("f,file", "Specify input or output file name", cxxopts::value<string>())
("h,help", "Show help")
```

Tk 前端如图：



4. 可改进处：

加设读缓存，可以加快读文件的速度，即压缩速度；

可以详细研究读写缓存大小以及编译器状况、硬盘状况对压缩、解压速度的影响并做出改进；

完善命令行参数，任意指定输入输出文件名称和解压目录，自定义输入输出缓存大小；

前段图形界面可以根据程序返回值和输出提供更友好的信息和提示。

5. 实验小结

本次试验 Huffman 算法本本身不是特别难，但涉及文件读写和字符处理，有一定难度，而可优化之处也很多，感觉考察了我各方面编程的综合能力。