Asst2 File Compressor

Zhizhang Deng, Xiaoxiao He

1. Methodology

In this program, we implement a lot of customized structs for increasing the overall performance of the program and reduce the memory usage.

1. Expandable (ExpandablePtr)

The data type Expandable is our implementation of list in Python, ArrayList in Java and vector in C++, which is a power tool for storing data when the size of the data is unavailable. Expandable stores an expandable char list and ExpandablePtr is an expandable array which can store various kinds of pointer. Then we have several APIs for this struct:

API name	Functionality
createExpandable(Ptr)	Create Expandable and initial the char array to 0
destroyExpandable(Ptr)	Free Expandable pointer and the char array
destroyExpandable(Ptr)WithoutFree	Free Expandable pointer only
expandExpandable(Ptr)	Expand Expandable array size
appendExpandable(Ptr)	Add one char to the Expandable char array
appendSequenceExpandable	Add one string to the Expandable
zeroUnusedExpandable	Set Expandable array to zero in specified index range

2. T-algorithm

When we do the pre-programming analysis of this project, we found an interesting algorithm in the book “The Art of Computer Programming” Vol.3 called T-algorithm, which has a time complexity of $O(n \log n)$. Because we need to count the occurrence of each “token” in the file, we need to use a high efficiency algorithm to search for the token. Regular linear search requires $O(n^2)$, which is enormous when dealing with a file that contains a lot of tokens (like a file that is a .bz2 file which has already encoded with Huffman algorithms, or any media file which has a lot of variance in the file). Therefore, we implement the algorithm for storing the values and find the token. We had implemented both linear search and T-algorithm. The benefit for using T-algorithm is enormous. With regular linear search to compress a MP4 file with the size around 5 MB is about 5 minutes, however, using the T-algorithm, we can reduce the time to within 5 seconds, which is a drastic change.

3. Huffman Tree

Because of the definition of the project, we need to implement a Huffman tree. Since we want the whole program to be faster, we decided to use a min heap to implement the Huffman tree. The overall performance of the process of Huffman Coding is $O(n \log n)$.

2. Design and Implementation

We redesign the Codebook format so that it is high efficiency and can handle binary nonsense.

The first line in the codebook will indicate how many lines (tokens) will be in the file. For each line, we have the Huffman codes followed by a tab. Then the first character following the tab is a number indicating whether the following string is normal token (1), or control code being rewritten into hexadecimal number (0).

Our implementation of the project accomplished 0 warning and 0 error during the compiling process and we have strict regulation of memory access so that we can eliminate random errors caused by improper memory access.

3. Time Complexity and Space Complexity Analysis

1. Time complexity:

It is easy to derive that the time complexity of our program is $O(n \log n)$ (n stands for the input size)

2. Space Complexity:

Since we actually need to store all information of the file into the memory, therefore, the space complexity is going to be $O(n)$.

4. Testing and Validation

We test our program by using our self-implemented python script, which will generate 100 folders contain 500 random files with the size ranging from $\frac{1}{64}$ MB to 8 MB. Then we will record the time for execution and the correctness of our program. We have tested our program with given python script and the result of the tests suggest that our program is robust not only to regular txt file which contains readable strings, but to random binary nonsense generated by /dev/urandom device.

Our program can handle ***500MB*** binary file on **iLab** machine (ilab3.cs.rutgers.edu) with ***6678199*** tokens (tested).

Size of the original file	Size of the compressed file	Size of the Codebook	Approx. Time used for compression	Memory used for compression
25M	200M	21M	24 Sec	486M
50M	401M	37M	40 Sec	866M
500M	4G	227M	460 Sec	6.55G

Table 1. Time and Memory Analysis for Compressing the Randomly Generated Nonsense Files