Practical Work 1

Algorithms II

Lz78 COMPRESSION AND DECOMPRESSION

FEDERAL UNIVERSITY OF MINAS GERAIS
Department of Computer Science
HENRY TAMEKLOE
2018500044
henrany@ufmg.br

Belo Horizonte - February 6, 2021

INTRODUCTION:

With the knowledge gained from class on some of the basic algorithm for pattern matching, it was upon us to practice the ideas gained into practice by using a trie for the compression and decompression of files using the lz78 compression and decompression algorithm. Trie is a very efficient data structure for storing string for consultation, reading and retrieval of words. Its consultation operation is very fast as the worst case of a search will take a maximum of O(length(text).

PROBLEM:

Files tends to be of string which when large can be difficult to store as they take a lot of space. There're a lot of algorithm for files compression and decompression but the problem at hand here is to use one of the know lz78 algorithm for the compression and decompression of the files. A brief explanation of the compression and decompression process is as follows:

The compression idea is to replace the occurrence of repeated words in a text. The algorithm initializes *last matching index* = 0 and *next available index* = 1. For each character of the input stream, the dictionary is searched for a match: {last matching index, character}. If a match is found, then *last matching index* is set to the index of the matching entry, and nothing is output. If a match is not found, then a new dictionary entry is created: dictionary[next available index] = {last matching index, character}, and the algorithm outputs *last matching index*, followed by *character*, then resets *last matching index* = 0 and increments *next available index*. Once the dictionary is full, no more entries are added. When the end of the input stream is reached, the algorithm outputs *last matching index.*

The decompression process is the reverse of the compression process as we retrieve even single character in the text. We do so by first traversing through the output stream and removing all the occurrence in index and placing them in as file for

DEVELOPMENT ENVIRONMENT

The whole algorithm using trie was built using the python programming language

To execute, you can run ***python3 filename.py*** . But for this work, we would like to execute each process differently
For the compression process:

```
python3 main.py -c file.txt  output.lz78
```

For decompression:

```
python3 main.py -x output.lz78 -o input.txt
```

DATA STRUCTURES AND ALGORITHM USED:

The data structure used here is the prefix tree which is also known as Trie. The Trie is an efficient data structure for storing and retrieving large text. The idea behind the data structure is to store each character of a word in a node with character, the code and also the if the character is the end of the word or not. Some of the basic operations which can be done is insertion into the trie, searching for a word and also deleting a word from the trie.

The pseudocode below is the implementation of the insertion and searching of a word in the trie

```
insertion(word):
    Root = Node()
    For key in word:
        If key not in root.children:
            root.children[key] = Node()
            root.code + 1
        Root = root.children[key]

Search(word):
    Root = Node()
    For key in word:
```

```
If char in node.children
    Root = root.children[key]
Else:
        Return false
Return true
```

This type of data structure in its own self is not space efficiency as some node may share the same prefix and we would want to store them in a single node.

IMPLEMENTATION:

Classes was used for the implementation of the node, trie, compress and decompress. It was chosen as a form of organizing our codes.

For the implementation, we first declared a Node function whereby each node will store the character, the code of that character and also whether it is the end of the word or not.
We then declared another function where we add a child to the trie if it doesn't exist. The class node has the methods Node(), which was used for the initialization and addChild(word,code) which was used to add the node to the tree.

For our class Trie, we used the following function as out methods, __init() where will set the head of the trie to the first node of the tree. The searchWord(word) will transverse through the tree and compare the characters of each of the nodes and see if the word exist. If the word exists, it returns true or else it returns false.

For our class Compress, we used the method Compressor for the compression of the files. We parametrized it with the input file and also the output files to which it will read and write respectively. We then we declared an empty variable to combine characters that we will be reading from the input file and also the code which will be incremented. We then transverse through the input file, and combine each character with the declared empty variable. We then search in our tree if the word exist or not. If it doesn't exist, we then add them to the tree and write it out with a in the output file beginning with the its code into the output file and increment the value of the code. If the word exist, we don't do anything. The output files was written in the form ***"0I0a0s10"*** as each number represent its code and each character is its text.

For our Decode class, we used the method Decompress to decompress the our file. We declared two variables combination which is an empty string, and also code. We then transverse through the input file, check whether the character is an number or not. If it is a number, we then add it to the tree with the word being the key of each of the nodes. We then write it out into the output stream and continue the process.

For the main, we called our two Classes, the compress and the decompress. The input and output files were read from the command line. All the functions were executed in the main.

TIME AND SPACE COMPLEXITY:

For the time complexity, each will analyzed each the functions and see which one of them takes more time and space and this will be our time complexity of the whole program.

Our search word takes a total of O(m) where m is the total number of words we would want to search in the tree. The add function takes O(1) as it just adds is to the tree if it is the end of the word. The compress compress and decompress takes a total time of O(n) where n is the total number of characters in the input text.

So analyzing all the the time complexities, we can conclude that, the whole program takes a total time O(n) time in the worst case to execute.

EXPERIMENTAL ANALYSIS:

Here we check the execution time of the the compression and decompression and the decompression.

Input	Size before compression	Size after compression	Rate
1	89 KB	90KB	0.01
2	197KB	195KB	0.009
3	924KB	798KB	0.008
4	1.2MB	1.2MB	0.0
5	173KB	172KB	0.0019
6	527KB	489KB	0.0092
7	58KB	62KB	0.01
8	248KB	252KB	0.01
9	144KB	144KB	0.00
10	52KB	55KB	0.01

From the table, it can be seen that, the compression process actually increases the size of files if the size for file is considerably small as the files with sized 52kb and 58 kb actually increased in size. But for files much bigger, the compression process seems to decrease it size as a proof of the file with size 527kb before compression became 489kb after compression.

CONCLUSION:

At the end we were able to compress and decompress a file using the lz78 algorithm. A lot was learnt in this project work as it gave me new insight on how to implement efficient algorithm for storing strings. With this algorithm, it will actually be safer to decrease the size of large files before storing them on some personal devices.

REFERENCES:

Introduction to Algorithms Cormen
https://www.geeksforgeeks.org/trie-insert-and-search/