Dictionary of Phrases Compressed

Riya Shrestha
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
rshrest2@stevens.edu

Rayhan Howlader
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
rhowlade@stevens.edu

***Abstract*—This project was designed to serve as a way to assess the learning in the Applied Data Structures and Algorithm class. For this project, our task was to read several books from the same genre and author to find the most common phrases. To accomplish that, the first task was to read the text and clean it to get rid of all the punctuation and spacing. Following that, we created a dictionary of words where the k most used words were stored in a hash table with their unique keys. Here all words are treated equally and no weight is assigned to them. Next, the task was to create a histogram of the most common phrases of the words found in the dictionary with the variation of only one word. After which all the phrases were encoded with keys of words from the dictionary and replaced all non-common words with the token 0. Lastly, the task was to create a data structure that stores all the phrases with a frequency higher than 1 and assigns a unique key to the phrase. Another program was also written to decode all the phrases. The books read in this project are the first 3 in the Harry Potter series authored by J.K. Rowling.**

***Index Terms*—hash table, histogram, dictionary, phrases, data structures**

I. Introduction

A dictionary of compressed phrases is a data structure that is used to store a mapping of data items to their compressed representations. In the C++ programming language, a compression dictionary can be implemented using a combination of data structures such as hash tables, trees, and arrays. The specific implementation discussed in this paper is by using hash tables for storing the application. This can be useful for reducing the amount of space required to store data, improving the speed of data transfer over a network, and reducing the overall cost. The overall goal is to provide efficient storage and retrieval of the compressed data. Similarly, there are several different algorithms to encode the words into text and find the sequences of the most common words, such as Huffman coding and LZW (Lempel-Ziv-Welch). The algorithm used in this project is similar to the LZW algorithm where unique codes are derived from the input.

A. Lempel-Ziv-Welch

LZW (Lempel-Ziv-Welch) is a lossless data compression algorithm that was developed by Abraham Lempel, Jacob Ziv, and Terry Welch in the 1980s. The LZW algorithm works by creating a dictionary of strings that it encounters as it processes the input data. When it encounters a string that is not in the dictionary, it adds the string to the dictionary and assigns it a unique code. The algorithm then outputs the code for the string and continues to process the input data. LZW compression can be used to compress any type of data, but it is particularly effective at compressing text-based data such as ASCII files or English language text. LZW has been widely used in a variety of applications due to its simplicity and efficiency. It has been adopted as a standard by the ISO and the ITU, and is used in many software programs and hardware devices.

B. Hash table

A hash table is a data structure that is used to store and retrieve data efficiently. It works by mapping keys to indices in an array, using a technique called hashing. To use a hash table, the keys are first transformed into hash values using a hash function. The hash values are then used to determine the index in the array where the key-value pair should be stored. This allows for fast insertion, deletion, and lookup of data, as the hash function enables the hash table to determine the index of the desired key-value pair in constant time. Hash tables are widely used in computer science, as they provide fast access to data and are efficient in terms of space and time. They are used in a variety of applications, including databases, caches, and symbol tables. There are several variations of hash tables, including open addressing hash tables, which store all key-value pairs in the same array, and chained hash tables, which store the key-value pairs in separate arrays that are chained together. The choice between these variations depends on the specific requirements of the application.

II. Problem Statement

At the beginning of the project timeline, the problem statement was defined as the following: Create an effective and efficient way to create, store, and encode a dictionary of phrases with the variation of one word. To accomplish this larger goal, the problem statement was broken down into smaller tasks.

1) Cleaning the initial text input to strip all the punctuation, extra spacing, and case.
2) Mapping all words found in the text with the count of the frequency into a dictionary and assigning unique ids to the top k words.
3) Constructing a dictionary of phrases that are made from the words in the dictionary of frequent words with at most 1 word not in the dictionary.

4) Replacing each word in the dictionary of phrases with its unique integer id as assigned in the dictionary of words.
5) Decoding the encoded dictionary of phrases.

III. Ideas and Design Process

The initial idea was to use a self-balancing binary search tree, map¡string, int¿ to create a dictionary of words. This dictionary would then be used to replace all the words not the dictionary with the special token 0. However, using the map¡string, int¿ implementation as the data structure for both the dictionary of words and phrases caused the time complexity of the project to be O(log(n)) on average.

The next attempt was to use a hash table as the main data structure for storing both dictionaries. Using unordered_map¡string, int¿ implementation for this compression project, decreased the time complexity to O(1) on average. Once storing the data structure was faster, the rest of the time was improved by using some common algorithms that would help in identifying patterns in the text and replacing those patterns with a unique key. The following steps were then taken to complete this project:

1) Cleaning the initial text input to strip all the punctuation, extra spacing, and case. This was done using the Regular Expression C++ library, regex.

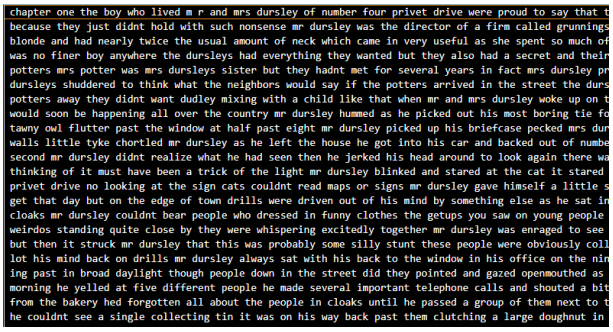


Fig. 1. Cleaned text file.

2) Mapping all words found in the text with the count of the frequency into a dictionary. From this dictionary, the k most frequent words were chosen and sorted from most frequent to least. Then the words were all mapped to a unique integer id from 1 to k, 1 being the most frequent and k being the least frequent. The mapping was done using the hash table implementation, unordered_map¡string, int¿.
3) Constructing a histogram count of the frequency of all phrases in the text, creating a dictionary of phrases that are made from the words in the dictionary of frequent words with at most 1 word not in the dictionary. The word that is not in the dictionary was replaced by the special token 0. This was completed using the .find() algorithm.
4) Replacing each word in the dictionary of phrases with its unique integer id as assigned in the dictionary of words. A new unordered_map¡string, int¿ was then created to

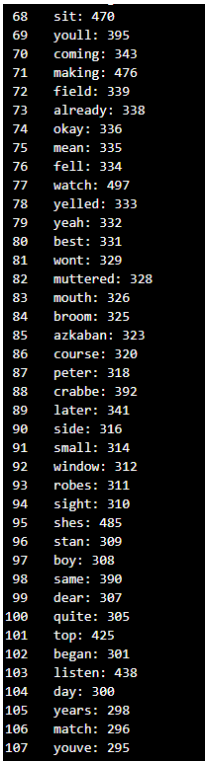


Fig. 2. Dictionary of the k most frequent words.

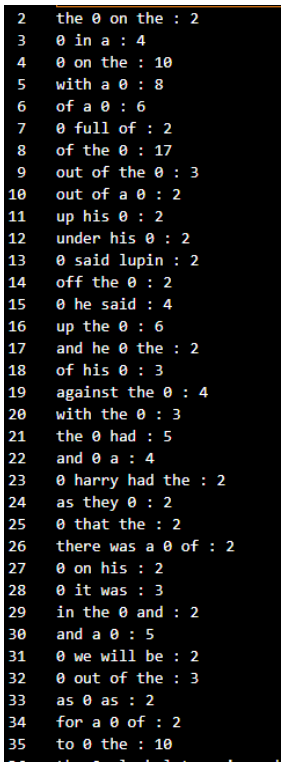


Fig. 3. Dictionary of the f most common phrases.

using the .insert() algorithm. In the new hash table, only phrases that appear with a frequency of more than f were

inserted.

```
20  26 4 0 5 : 228
21  11 0 1 : 195
22  5 59 0 : 193
23  5 0 8 : 192
24  3 0 2 : 190
25  2 4 50 0 : 189
26  12 0 1 : 188
27  0 27 5 59 : 187
28  2 0 32 201 : 186
29  0 193 2 : 185
30  7 0 11 : 179
31  0 37 4 : 178
32  5 9 0 163 : 176
33  16 1 0 : 175
34  61 9 0 : 174
35  0 6 6 : 183
36  0 5 34 : 173
37  39 8 219 0 5 : 172
38  2 1 0 5 : 171
39  223 1 0 : 170
40  1 0 29 : 167
41  2 0 28 9 : 177
42  21 94 0 15 : 166
43  28 2 49 1 0 : 164
44  11 8 0 : 165
45  0 182 239 : 163
46  1 0 60 : 231
47  18 1 0 1 : 162
48  3 9 0 : 160
49  237 1 0 : 158
50  0 10 6 : 156
51  0 90 7 : 152
52  5 0 85 : 150
53  5 0 18 1 : 148
54  12 1 0 2 : 146
55  0 266 293 : 144
56  0 58 118 31 : 233
57  0 2 24 29 : 143
```

Fig. 4. Dictionary of the encoded phrases.

5) Decoding the encoded dictionary of phrases was the last task covered in the scope. Another program was created that read both dictionaries and decode all the unique integers back to their respective words.

The scope of this project does not include re-writing the text with the unique integer ids of the words and the phrases. The generic logic behind the code can be seen in the figure below.

IV. Results and Analysis

After completing the code portion of the project, it was found that using hash implementation, unordered_map increased the speed in which the program ran in comparison to using the self-balancing binary search tree implementation, map. The figure below shows the output from the terminal. It can be seen that the time elapsed including the time it takes the user to input the two integers is 21.209 secs.

The purpose of the project was to learn about data compression using data structures and algorithms. Since the encoded phrases were not re-written into a text file, a proper analysis of the entire text file being compressed could be done. The only analysis that can be conducted is by looking at the file size of the dictionary of phrases in 3 different stages:

1) When the phrases are initially created
2) When the phrases are encoded with their word-specific keys
3) When the phrases have been decoded

```
10  the 0 of a : 586
11  very 0 in the : 625
12  and 0 and : 682
13  along the 0 : 590
14  of magical 0 : 694
15  said hagrid 0 : 617
16  he said 0 : 668
17  lupin 0 the : 584
18  for his 0 : 582
19  0 who was : 627
20  0 from the : 574
21  with 0 and : 572
22  with 0 of : 587
23  0 is a : 610
24  0 out of the : 561
25  0 from his : 629
26  0 as a : 592
27  0 in the : 502
28  but a 0 : 613
29  the 0 a : 570
30  and 0 that : 601
31  by the 0 : 567
32  to 0 a large : 721
33  up the 0 : 527
34  0 professor trelawney : 562
35  got a 0 : 596
36  of the 0 in the : 568
37  to the 0 : 514
38  0 into the : 524
39  with the 0 : 566
```

Fig. 5. Dictionary of the decoded phrases.

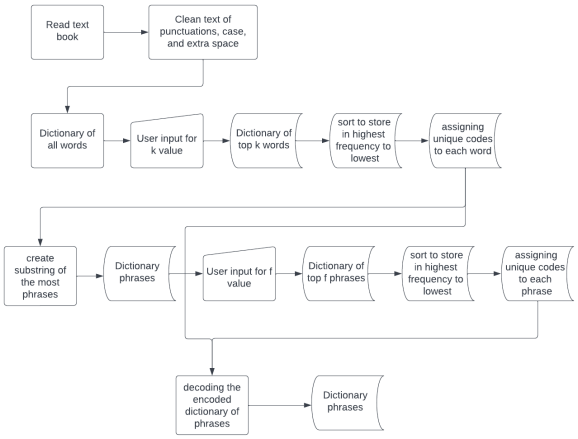


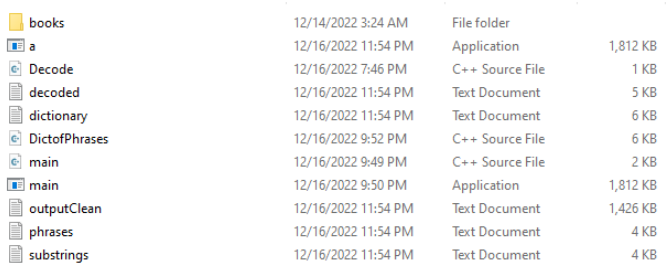
Fig. 6. Flow chart of how the code is processed.

```
PS C:\Users\rio_s\Documents\Fall22\CPE593\DictionaryofPhrases\source> g++ main.cc
PS C:\Users\rio_s\Documents\Fall22\CPE593\DictionaryofPhrases\source> .\a.exe
Done cleaning 3 Harry Potter books.
Choose the k most common words to be in the dictionary: 100
Done creating dictionary.
Choose the f most common phrases to be in the dictionary: 12
Done finding common phrases.
Done decoding phrases.
Elapsed time: 21.309 seconds
PS C:\Users\rio_s\Documents\Fall22\CPE593\DictionaryofPhrases\source>
```

Fig. 7. Output from the terminal.

It is interesting to note that the file size did not change from

when the phrases were initially created to when they were encoded with the word-specific keys. However when the file was decoded the file size increased by 1 kb.



books	12/14/2022 3:24 AM	File folder	
a	12/16/2022 11:54 PM	Application	1,812 KB
Decode	12/16/2022 7:46 PM	C++ Source File	1 KB
decoded	12/16/2022 11:54 PM	Text Document	5 KB
dictionary	12/16/2022 11:54 PM	Text Document	6 KB
DictofPhrases	12/16/2022 9:52 PM	C++ Source File	6 KB
main	12/16/2022 9:49 PM	C++ Source File	2 KB
main	12/16/2022 9:50 PM	Application	1,812 KB
outputClean	12/16/2022 11:54 PM	Text Document	1,426 KB
phrases	12/16/2022 11:54 PM	Text Document	4 KB
substrings	12/16/2022 11:54 PM	Text Document	4 KB

Fig. 8. Note the sizes for substrings, phrases, and decoded files.

V. Conclusion

We conclude the paper by citing that Word compression is a technique used to reduce the size of a document or file by removing redundancies and finding ways to represent the same information using fewer bits. This can be useful for storing large volumes of text or transmitting documents over the internet, where smaller file sizes can lead to faster transfer times. The method used was done through a combination of Lempel-Ziv-Welch and hash tables. By using this data structure, especially using hash maps the program had run much better than the previous interactions made. This is because hashmaps don't use up much time no matter the size of the set due to the complexity being O(1). Overall, word compression can effectively reduce the size of text-based documents, but the amount of compression achieved will depend on the characteristics of the text and the method used.

References

[1] C++ Unordered$_{m}apLibrary - find()Function.(n.d.).https : //www.tutorialspoint.com/cpp_standard_library/cpp_unordered_map_find.htmGeeksforGeeks.(2018a, September19).unordered_mapend()functioninC +STL.https : //www.geeksforgeeks.org/unordered_map - end - function - in - c - stl/

[2] GeeksforGeeks. (2021, November 8). LZW (Lempel–Ziv–Welch) Compression technique. https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique

[3] GeeksforGeeks. (2022, October 26). Huffman Coding — Greedy Algo-3. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

[4] DevPal. (2021, February 17). LZW Compression Algorithm Explained — An introduction to data compression. YouTube. https://www.youtube.com/watch?v=KJBZyPPTwo0

[5] GeeksforGeeks. (2020b, May 31). Sorting a Map by value in C++ STL. https://www.geeksforgeeks.org/sorting-a-map-by-value-in-c-stl/

[6] GeeksforGeeks. (2022b, November 9). map vs unordered$_{m}apinC ++.https : //www.geeksforgeeks.org/map - vs - unordered_map - c/GeeksforGeeks.(2021a, September26).unordered_mapat()inC ++.https : //www.geeksforgeeks.org/unordered_map - at - cpp/$

[7] GeeksforGeeks. (2021c, November 29). Vector of Unordered Maps in C++ with Examples. https://www.geeksforgeeks.org/vector-of-unordered-maps-in-c-with-examples/

[8] GeeksforGeeks. (2018c, December 19). unordered$_{m}apinsertinC ++STL.https : //www.geeksforgeeks.org/unordered_map - insert - in-c-stl/GeeksforGeeks.(2022a, September9).vectorinsert()functioninC++STL.https : //www.geeksforgeeks.org/vector - insert - function - in - c - stl/$

[9] Abdul Bari. (2018, February 8). 3.4 Huffman Coding - Greedy Method. YouTube. https://www.youtube.com/watch?v=co4$_{a}hEDCho$