COSC 320 – 001

Analysis of Algorithms

2022/2023 Winter Term 2

Project Topic Number: 1

Project Second Milestone (Group 32)

Keyword Replacement in Corpus

Group Lead: Anitej Isaac Sharma

Group Members:

Anitej Isaac Sharma

Abdirahman Hajj Salad

Yuki Isomura

Abstract

This is the Second Milestone for our Keyword Replacement (in Corpus) Algorithm. In this milestone, we succeeded in coming up with the Algorithm Analysis, Data Structure, Unexpected Cases/Difficulties, and Task Separation and Responsibilities.

Algorithm Analysis

Given a Corpus of documents *C*, a list of keywords *K*, and their corresponding phrases *P*, we aim to find all occurrences of keywords in *K* in documents in *C* and replace them with the appropriate corresponding phrases in *P*. Let *C = {c1, c2, c3, ..., cd}* be the corpus of documents, *K = {k1, k2, k3, ..., kn}* be the list of keywords, and *P = {p1, p2, p3, ..., pn}* be the corresponding phrases, *d* is the number of documents in *C* and *n* is the number of keywords and their corresponding phrases.

Refined Algorithm B (HashMap Implementation):

- Uses a HashMap to store *K* and *P*, where *K* are the keys and *P* are the values. For example, entries of the HashMap would look like *{k1, p1}, {k2, p2}, {k3, p3}, ..., {kn, pn}*, where the elements marked by *k* are the keywords being used as keys and the elements marked by *p* are the corresponding phrases being used as values.
    - Note that this is a part of pre-processing which takes *O(n)* time in the worst case. However, this won't affect the overall time complexity of the algorithm as it only needs to be done once.
- This algorithm, first, goes through *K* and *P* and stores each pair of elements in the HashMap. Iterating through *K* and *P* simultaneously, which doesn't require extra time since the number of keywords and phrases are the same, takes *O(n)* time. Inserting each pair of elements into the HashMap takes *O(1)* time.
- Then, it takes the corpus of documents, goes through each one of them, and for each document, it goes through all the words and searches for them in the HashMap. Iterating through the corpus of documents takes *O(d)* time and iterating through each word within a document takes *O(w)* time.
- When searching for a keyword in the HashMap, the algorithm performs a hash function on the keyword to determine its position within the HashMap, allowing for *O(1)* retrieval of the corresponding phrase. If there exist multiple keywords with the same hash values or at the same index, a linked list can be used to store all of those keywords, which would increase the search time to *O(k)*, where *k* is the number of elements in the linked list.

The overall worst-case time complexity of Refined Algorithm B (HashMap Implementation) is ***T = O(n+d*w)*** and the space complexity is ***S = O(n+d)***.

```
#Pre-Processing
  Create an empty HashMap
  FOR each pair of elements (k, p) in K and P
      Insert the pair (k, p) into the HashMap

  FUNCTION ALGO_B_REFINED:
   FOR each document d in C
       Split d into words
       FOR each word w in d
          IF the last character of w is a period (.)
             Split w from the period and store w in a new variable, str
          ELSE
             Set str = w
             Calculate the hash value for str using a hash function
             IF the hash value exists in H
                IF there exists a linked list of elements (containing
the element we're searching for) at the index
                   search for the element in the linked list and replace
the word in d with the corresponding phrase in P
                ELSE IF str is the keyword stored at the index in H
                   replace the word in d with the corresponding phrase
in P
                ELSE
                   Move to the next word in d
   RETURN the updated list of documents with keywords replaced by their
corresponding phrases
```

Analysis:

- Refined algorithm B only calculates the hashcode for a particular word and uses that to search through the list of keywords which, as per the functionality of a HashMap, takes O(1) expected time and O(n) in the worst case, assuming that there are collisions and linked lists are used to hold multiple keywords at an index.
- Pre-processing of the algorithm, which involves constructing the HashMap, takes O(n) time in the worst case. This only needs to be done once, so it won't affect the overall time complexity of the algorithm.
- The use of a HashMap to store the keywords and phrases improves the efficiency of the algorithm by providing constant-time access to the corresponding phrases for each keyword. However, there could be possible collisions during pre-processing and these are a few methods of dealing with them:
    - One approach to handling collisions is to use separate chaining, where each HashMap entry where the collisions occur now contains a linked list of elements that have the

same hash value.
        - In this case, the time complexity of the algorithm can increase to O(k), where k is the number of elements in the linked list.
    - The other better solution is holding another HashMap as a value of collided keys. The hashcode of this inner-HashMap is calculated based on other equations as the outer-HashMap so that they can avoid the same collision. By doing that, the time it takes to search keywords within the same key of outer-HashMap is better than O(m). If the total number of keywords was massive, we may have to do this process recursively until only 1 element remains in the same key of the multiple inner-HashMaps.
    - It is possible, in such a case, that if some keywords keep making collisions forever in the multiple inner-HashMap, the program runs into an infinite loop (For this to occur those keywords have to keep colliding even with different hashcodes).

Proof of Correctness:

- *Loop invariant:* At the start of each iteration of the outer loop, all the keywords in K that were found in the reviewed documents have been replaced with their corresponding phrases in P.
- *Initialization:* Before the first iteration of the outer loop, there have been no keywords that were found in any of the documents, so the loop invariant holds trivially.
- *Maintenance:* Assuming that the loop invariant holds at the start of an iteration of the outer loop. Then, we consider an iteration of the inner loop that searches for keywords in the current document. If a keyword is found in the current document, the loop replaces the keyword with its corresponding phrase in P, and the loop invariant is still true at the end of this iteration of the inner loop. If a keyword is not found in the current document, the loop invariant is still true since no replacements have been made.
- *Termination:* When the outer loop is terminated, all documents in the corpus have been searched for keywords, and all keywords have been replaced with their corresponding phrases in P. Therefore, the loop invariant holds after the algorithm terminates.

Since the loop invariant holds before the first iteration, during the loop, and after the loop terminates for Refined Algorithm B, it is correct.

Data Structure

Choice of the Data Structure and Justification:

- Our choice of data structure is a HashMap which we will implement to search for a keyword, based on the current word, in our dictionary of abbreviations and replace it with its full form once a match is found. Otherwise, move on to the next word in the document.
- Using a HashMap makes it possible to search with an expected run-time of O(1), which is more efficient than Algorithm A (Brute-Force) which goes through a list of keywords with

O(n) and O(m) for each word in the document to look for a match, where n is the number of keywords and m is the length of the current word/keyword.

- Even if all keywords collide, searching for an item within the same slot of keywords takes O(n) as the keywords would then potentially be stored using separate chaining. This indicates that Algorithm B is faster and more efficient than Algorithm A.
- As algorithm B needs to store a HashMap that holds all the keywords, space complexity is O(n). However, Algorithm A needs to store the list of keywords and the list of corresponding phrases to find a match. Each of those lists also holds the space complexity of O(n). Thus, the efficiency in memory consumption does not change by switching the algorithm. Moreover, since the size of lists of keywords is relatively smaller than the size of files that hold original phrases, it is expected to have no StackOverflow Error unless the size of the documents is too massive.

Unexpected Cases/Difficulties

- Input Format Uncertainty (ex. tweets can be in different languages)
- Typographical Errors (ex. multiple periods at the end of the sentence or misspelled words)
- For specific contexts where a word in a document is not supposed to be an abbreviation but it is in the list of keywords, it would get replaced with a corresponding phrase by the algorithm.
- Numerous collisions in HashMap initialization in algorithm B. As the number of collisions increases, search time complexity gets worse. This could lead to Infinite collisions (a rare phenomenon) for which we implement a recursive HashMap.
- If the size of the corpus is very large, the memory usage of the program may become an obstacle as all the documents are stored in the memory. This could lead to slower performance, however, one way to address this issue is to store the documents on disk and only load them into memory as needed.

Task Separation and Responsibilities

Anitej Isaac Sharma Algorithm Analysis & Unexpected Cases/Difficulties	**Yuki Isomura** Data Structure	**Abdirahman Hajj Salad**