COSC 320 – 001

Analysis of Algorithms

2022/2023 Winter Term 2

Project Topic Number: 1

Project First 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 First Milestone for our Keyword Replacement (in Corpus) Algorithm. In this milestone, we succeeded in coming up with the problem formulation, pseudo-code, algorithm analysis, unexpected cases/difficulties, and task separation and responsibilities.

Problem Formulation

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 $C=[c_1,c_2,c_3,.....,c_d]$ be the corpus of documents, $K=[k_1,k_2,k_3,....., k_n]$ be the list of keywords, and $P=[p_1,p_2,p_3,.....,p_n]$ be the corresponding phrases.

The Naïve Algorithm A (Brute-Force Search) iterates through C, and for each document in it, it goes through all the words, and compares them with all the keywords in K.

- Iterating through C takes $O(d)$ time, where d is the number of documents in the corpus.
- For each document, iterating through it takes $O(w)$ time, where w is the number of words within each document.
    - Here, if we come across a word followed by a period (full stop or end of the sentence), we need to split the word and the period, which takes linear time as per the number of periods within each document.
- Traversing through the words in each document, we're gonna compare each word with each keyword in K to look for equal lengths, which takes $O(n*w)$ time in the worst case, where n is the number of keywords, or in other words, n is the length of K and P.
- If a match is found, we then compare the current word with the current keyword, letter by letter, which takes $O(m)$ time, where m is the length of the current word/keyword.
    - If all the letters match, we'd then replace the word in the document with the keyword's corresponding phrase in P which always takes $O(1)$ time.

Putting it all together, the overall worst-case time complexity is $T=O(d*w*n*m)$, and the space complexity is $S=O(n)$.

Our goal is to use Refined Algorithm B (HashMap Implementation) to store K and P, where K are the keys and P are the values. For example, entries of the HashMap would look like $\{k_1,p_1\}$, $\{k_2,p_2\}$, $\{k_3,p_3\}$, ..., $\{k_n,p_n\}$, where the elements marked by k are the keywords being used as keys and the elements being marked by p are the corresponding phrases being used as values.

For this algorithm, we first go through K and P and store each pair of elements in the HashMap.

- Iterating through K and P simultaneously, as they would be of the same length with corresponding values, takes $O(n)$ time.
- As per the functionality of a HashMap, inserting each pair of elements into the HashMap always takes $O(1)$.

****This is part of the pre-processing, so it won't be an iterative process like the one mentioned below and won't be directly accountable for the overall time complexity of the algorithm.*

Then, we take the corpus of documents, go through each one of them, and for each document, go through all the words, and search for them in the HashMap.

- Iterating through the corpus of documents takes $O(d)$ time.
- For each document, Iterating through it takes $O(w)$ time.

- ◦ Here, if we come across a word followed by a period (full stop or end of the sentence), we need to split the word and the period, which takes linear time as per the number of periods within each document.
- As we go through the words in each document, we're going to perform a search in the keys section of the HashMap for each word to look for a match, which, as per the functionality of a HashMap, takes *O(1)* expected time and *O(n)* for the worst-case if a loop through the entire HashMap is required.
  - ◦ Note that the search is based on comparing the hashcode of the word and of a keyword in *K*, and if they appear to be equal, the keyword is returned.

Lastly, if we find a match, our job is to replace that keyword in the document with the corresponding phrase stored in the HashMap.

- Replacement of keywords by their corresponding phrases always takes *O(1)*.

Putting it all together, the overall expected time complexity is $T=O(n+d*w)$, *the* worst-case time complexity would be $T=O(n+d*w*n)$, and the space complexity would be $S=O(n)$.

Pseudo-Code

Naïve Algorithm A (Brute-Force Search):

```
FUNCTION brute_force_keyword_phrase_search(C, K, P) { // C = corpus of documents, K = list of keywords, P = list of corresponding phrases
  FOR i = 0 TO C.length   // go through each document in the corpus
    FOR j = 0 TO C[i].length   // go through all words in each document
      word = C[i][j]    // get the current word
      IF word[word.length-1].equals(".")    // if the word is followed by a period, split the word and period
        word = word.split(".")[0]
      ENDIF
      FOR k = 0 TO K.length
        IF P[k].length == word.length
          counter = 0
          FOR l = 0 TO word.length
            IF word[l] == K[k][l]
              counter += 1
            ENDIF
          ENDFOR
          IF counter == word.length || counter == K[k][l].length
            phrase = P[k]
            C[i][j] = phrase
          ENDIF
        ENDIF
      ENDFOR
    ENDFOR
  ENDFOR
  RETURN C
}
```

Refined Algorithm B (HashMap Implementation):

```
FUNCTION keyword_phrase_algorithm(C, K, P) {  // C = corpus of documents, K = list of keywords, P = list of corresponding phrases
  hash_map = HashMap()    // create an empty HashMap
```

```
  FOR i = 0 TO K.length      // store each pair of elements from K and P into the HashMap
    hash_map.put(K[i], P[i])
  ENDFOR
  FOR i = 0 TO C.length      // go through each document in the corpus
    FOR j = 0 TO C[i].length     // go through all words in each document
      word = C[i][j]      // get the current word
      IF word[word.length-1].equals(".")     // if the word is followed by a period, split the word and period
        word = word.split(".")[0]
      ENDIF
      IF hash_map.contains_key(word)    // search the HashMap for the current word
        phrase = hash_map.get(word)       // replace the keyword with the corresponding phrase
        C[i][j] = phrase
      ENDIF
    ENDFOR
  ENDFOR
  RETURN C
}
```

Algorithm Analysis

The major differences between Naïve Algorithm A and Refined Algorithm B are the time complexities of the search for keywords.

- ➔ In algorithm A, for a specific word, brute-force goes through the list of keywords and compares the lengths. If there is a keyword of equal length to that of the current word, then it would compare the two, letter by letter. All of this takes O(n*m) time.
- ➔ On the contrary, 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.

Even in the worst case, assuming all the keywords are in the same key of HashMap, in other words, all keywords made a collision, it takes O(n) to go through all the keywords assuming comparing each keyword takes O(1). If we assume that we use Brute-Force search in comparing each keyword within the same key of HashMap, it takes O(n*m) maximum. This implies that algorithm B has at least the same time complexity as algorithm A and will be better as more keywords are stored in HashMap without collisions.

Another difference is the preprocessing of HashMap. In this process, we need to deal with collisions within HashMap. One of the solutions is to contain a list of collided keywords as a value of HashMap and use Brute-Force search to look through as it is mentioned above. 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 is stored in the same key of the multi-inner-HashMap. In this case, we must remember that if some keywords keep making collisions forever in the multi-inner-HashMap, the program runs into an infinite loop. (This is nearly impossible to happen because those keywords have to keep making collisions with different hashcode calculations to reproduce this phenomenon)

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)
- ➔ Massive number of collisions in HashMap initialisation in algorithm B. (As the number of collusion increases, search time complexity gets worse but the frequency of collusion is an uncertain variable.)
- ➔ Infinity collisions in case we implement a recursive HashMap. (Nearly impossible to occur)

Task Separation and Responsibilities

Anitej Isaac Sharma	**Yuki Isomura**	**Abdirahman Hajj Salad**
→ Problem Formulation → Pseudo-Code	→ Algorithm Analysis	→ Unexpected Cases/Difficulties