COSC 320 – 001
Analysis of Algorithms
2022/2023 Winter Term 2

Project Topic Number: #1

Keyword Replacement

Group Lead:

Sanjith Senthil

Group Members:

Issa Hashim, Cole Van Steinburg, Sanjith Senthil

Abstract

In this milestone, we developed a more efficient algorithm to the keyword replacement problem using a hashmap / dictionary. We considered different edge cases to make the algorithm robust and be able to accept multiple bodies of text. We analyzed the time and space complexity of the algorithm and performed a proof of correctness of the algorithm.

Problem formulation

Given an array of strings 'paragraphs' containing n total words, and an array of tuples 'abbreviations' of length m, where each tuple is of the form (abbreviated word, replacement word). The algorithm should modify the array 'paragraphs' such that all abbreviated words in 'abbreviations' that appear in 'paragraphs' are substituted with their corresponding replacement words.

Here's an example:

```
paragraphs = [

      "Imho, I think that the earth is flat. They should take a trip to the
       edge and see for themselves asap." ,

      "He has been in the industry for 20 yrs."

]

abbreviations = [ ("yrs", "years"), ("imho", "in my humble opinion"), ("CEO",
"Chief Executive Officer"), ("asap", "as soon as possible") ]
```

After running the algorithm, paragraphs would be updated to:

```
paragraphs = [

      "In my humble opinion, I think that the earth is flat. They should take
       a trip to the edge and see for themselves as soon as possible." ,

       "He has been in the industry for 20 years."

]
```

Pseudo-code

```
# paragraph_list is an array of paragraphs where each paragraph is a string.
paragraph_list = Read_Paragraph_List()

# replacement_list is a list of tuples where each tuple is of the form (abbreviated word, replacement word)
replacement_list = Read_Replacement_List()

# replacement_dict is a mapping of abbreviations to full forms in the form "word to replace" : "replacement"
#     Example of replacement dict: {
#                                       "imho": "in my humble opinion",
#                                       "can't": "can not",
#                                   }
replacement_dict = {replacement_pair[0] : replacement_pair[1] for replacement_pair in replacement_list}
```

```
# An array storing common punctuation symbols.
punctuation_list = [",", ".", "!", "?"]

for paragraph_index, paragraph in enumerate(paragraph_list):
        word_list = paragraph.split()

        for word_index, word in enumerate(word_list):

                # Split the punctuation from the word itself
                punctuation = ""
                while word[-1] in punctuation_list:
                        # Add the character to punctuation string
                        punctuation += word[-1]
                        # Remove the character from word string
                        word = word[:-1]
                # Flip punctuation string as they were read backwards
                punctuation = punctuation[::-1]

                # Check if the word needs to be replaced
                try:
                        replacement_word = replacement_dict[word.lower()]
                        # Capitalize first letter if it was capitalized in initial text
                        if word[0].isupper():
                                word_list[word_index] = replacement_word.capitalize() + punctuation
                        else:
                                word_list[word_index] = replacement_word + punctuation

                except KeyError:
                        pass

        # Add the fixed word list back to your list of paragraphs
        paragraph_list[paragraph_index] = " ".join(word_list)

# Print results
for paragraph in paragraph_list:
        print(paragraph)
```

Algorithm Analysis

Data Structure: We used a hashmap, which is also known as a dictionary in Python, to make all comparisons to the abbreviation list O(1) rather than O(m). We can make this tradeoff to move some of the time complexity of the comparisons to auxiliary space complexity. This brings time complexity from O(n*m) to O(n) and auxiliary space complexity from O(1) to O(m). However, the total space complexity will still remain the same, that is O(n + m). Note that n is the total number of words in the paragraphs array and m is the length of the abbreviations list.

Time complexity: With the improved approach using a hashmap, that is for each word in the paragraph, the algorithm checks a dictionary using the word as a key to retrieve a replacement word and replaces them. The time complexity is O(n) where n is the total number of words in the array of paragraphs.

Proof of running time:

Operations performed:

- Split paragraph into a list of words (done once for each paragraph) | *n / number of paragraphs operations*

- Loop through the following for each of the n words | *n times inner operations*
    - Separate the punctuation from the word itself | *k operations (constant)*
        - Loop backwards through characters in the word, checking if each is a punctuation character.
        - If it is, add it to the punctuation list and remove it from the word variable.
    - Do a lookup of the word in the replacement hashmap | *k operations (constant)*
    - If the word is in replacement hashmap, replace it with the unabbreviated form and add back punctuation | *k operations (constant)*
    - If the word is not in replacement hashmap, move onto the next word | k operations (constant)
- Join the new list of words into a single string to save the new unabbreviated paragraph. | *n / number of paragraphs operations*

From our operations list, we see that we have:

$$\begin{aligned} & n/(\#\text{ paragraphs}) + n/(\#\text{ paragraphs}) + n*(k+k+k+k) \\ &= 2n/(\#\text{ paragraphs}) + n*4*k \\ &= (2/\#\text{ paragraphs} + 4k) * n \\ &= c * n && \text{// c is a constant} \\ &= O(n) \end{aligned}$$

Space complexity: The total space complexity is O(n + m), where n is the total number of words in the paragraphs array and m is the number of elements in the abbreviation list. The auxiliary space complexity is O(m) as the algorithm uses additional space for storing everything from the abbreviation list into an abbreviation hashmap.

Proof of correctness:

- Loop invariant: Each paragraph will be split into an array of words called A. At the start of each iteration of the while loop, the array A[0 .. i-1] contains the i - l words with all abbreviations replaced.

- Inductive Hypothesis: All abbreviations from indices [0 .. k] have been substituted with their replacement word.

- Base case (i = 0): By definition, a paragraph of 0 words will not contain any abbreviations.

- Inductive step: Assume the inductive hypothesis is true (all abbreviations from indices [0 .. k] have been replaced with their unabbreviated form), then we will show that the algorithm holds for iteration k+1. The single word at index k+1 will be checked against the abbreviations in the replacement hashmap B. If the word is found, it will be replaced by the unabbreviated form. If the word is not found, then it is already in its unabbreviated form. Thus, after running the algorithm it is guaranteed that there are no abbreviations from elements [0 .. k+1] and the loop invariant holds.

- Conclusion: At the termination of the algorithm, all abbreviations in A has been replaced.

Unexpected Cases/Difficulties

Case sensitivity: The case of the abbreviated words in the input array needs to be preserved when modified. For example, ‘Imho’ in the input array should be replaced with ‘In my humble opinion’. Similarly, ‘imho’ should be replaced with ‘in my humble opinion’. To achieve this, when comparing the acronym, each individual word in the input array is compared to the abbreviation list in lowercase; but when modifying the input array, we check the case of the first letter in the abbreviated word and perform the replacement accordingly.

Punctuation symbols: When comparing each word in the input array to those in the replacement list, punctuation can make the equality check fail. For example, “lol!” would not match “lol”. To fix this issue, we first separated the punctuation from the word before performing the comparison, then added punctuation back after.

Task separation and responsibilities

We divided the work among us equally, meaning that all members of our group contributed to every task of the milestone together. We collaboratively worked on the algorithm design, analysis, proofs, and the implementation of pseudo code. In addition, Cole and Issa were responsible for setting up the group meeting and creating the document, and Sanjith was responsible for communicating with the instructor and TA’s.