COSC 320 – 001

Analysis of Algorithms

2022/2023 Winter Term 2

Project Topic Number: 2 Milestone 3

KML and LCSS

Group Members: Samuel S, Samuel H. Jimmy J.

Problem Formulation.
LSCC Algorithm
Input:

- A corpus of n documents D = {d1, d2, ..., dn} in plain text format
- A potentially plagiarized document P in plain text format
- An integer parameter k representing the length of substrings to compare
- A similarity threshold s

Output:
The set of documents D that P was plagiarized from

```
1. For each document di in D:
1.1 Create a list Li of all substrings of length k in di
2. Create a list Lp of all substrings of length k in P
3. For each substring sp in Lp:
3.1 For each document di in D:
3.1.1 For each substring si in Li:
3.1.1.1 Use the LSCC algorithm to find the length l of the longest common subsequence between sp and si
3.1.1.2 If l/k >= s, add di to the set of matched documents
4. Return the set of matched documents
```

Such that for each document di present, there exists a pattern of 3 or more consecutive words in A that match a pattern of 3 or more consecutive words in document di with at least 24% consecutive words matched of the word length of di

KMP Algorithm
Input:

- A corpus of n documents D = {d1, d2, ..., dn} in plain text format
- A potentially plagiarized document P in plain text format

Output:
The set of documents from which P was plagiarized, represented as a set S

```
1. Define a function KMP(pattern, text) that implements the KMP algorithm to find all occurrences of a pattern in a given text. The function should return a list of indices where the pattern occurs in the text.
2. Initialize an empty set S
3.For each document di in the corpus:
a. Find all occurrences of p in di using the KMP algorithm.
b. If any occurrences are found, add di to the set S.
4. Return S
```

Such that for each document di present, there exists a pattern of 3 or more consecutive words in A that match a pattern of 3 or more consecutive words in document di with at least 24% consecutive words matched of the word length of di

Pseudo-code.
for both

```
import documents(corpus_file_directory), ppdoc(potentially_plagarized_document)
number_of_docs_in_corpus = 0
```

```
for i in range (length(documents)):
    for j in range(length(documents[ i ].comment)):
        number_of_docs_in_corpus += 1            //this is to count the number of separate
                                                 //comments so that we can count each //comment as a
                                                 separate document as //checking plagiarism between
                                                 2 separate //comment fragments makes no sense

corpus = [String s = “” for i in range(length(documents))]      // blank string for every comment
n = 0
for i in range (length(documents)):
    for j in range(length(documents[ i ].comment)):
        for k in range(length(document[ i ].comment[ j ].letters)):
                letter = letter.lowercase
                if letter == a,b,c,...x,y,z + “ ”:
                        corpus[n] = corpus[n] + document[ i ].comment[ j ].letters[ k ]
    n += 1                                       //counter to keep track of which document we
                                                 //are on
String ppdw = “”                                 //ppdw potentially plagiarized document words
wc = 0                                           //wc = word count in ppdw
for i in range(length(ppdoc)):
  if(ppdoc[ i ].lowercase() == a,b,c, …, x,y,z, “ ”):     //this gets rid of the potential punctuation or
        ppdw = ppdw+ppdoc[ i ].lowercase()                //capitalization problems that could occur
        if((ppdoc[ i ] == “ ” and ppdoc[ i +1] != “ ”) or ppdoc[ i+1 ] == None):
                wc = wc +1
count = 0                                        //number of plagiarized words found
tc = 0                                           //tc for temporary count, series of words are only
                                                 //counted if  >=3 words in a row are potentially
                                                 //plagarized
perc = 0                                         //%of words plagiarized
```
psudo for KMP
```
for i in range (length(corpus)):
    KMP-MATCHER(corpus, ppdw)
perc = (count/wc)*100
if(perc >= 24):
  print(“the document in question has a high probability of having been plagarized showing ” + perc + “% of the
  document which is potentially plagarized”)
else:
  print(“this document is not likely plagarized from the corpus provided, it shows a potential” + perc + “%
plagarized material which is within tolerance”)
KMP-MATCHER(comment, ppdw):
  n = comment.length
  m = ppdw.length
  e = PREFIX-FUNCTION(ppdw)
  q = 0
  for i in range(1,n):
        while(q>0 and ppdw[q+1] != comment[i]):
                temp = q
                q = e[q][0]
                prefix_spaces = e[temp][1]
                tc = tc - prefix_spaces                  //temporary count = temporary count - number of
                if:(tc >= 3):                            //words that are in the longest prefix that are also a
                                                         //suffix (avoids double counting)
                    count = count + tc
                    tc = 0

                tc = tc + prefix_spaces                  //adds back words that were temporarily moved out
                                                         //of tc
            if ppdw[q+1] == comment[i]:
                if(comment[i] == " " or None):                   // then we are at the end of a word
                    tc = tc +1
                q=q+1
            if(q>0 and ppdw[q+1] != comment[i]):
                if(tc >= 3):
                    count = count + tc
                    tc = 0
   if(tc >= 3):
        count = count + tc
        tc = 0


PREFIX-FUNCTION(ppdw):
   m = length(ppdw):
   e = [[0 for i in 2] in range m]
   k = 0
   spaces = 0                                            // counts number of words
   for q in range(2,m)
        while(k > 0 and ppdw[k+1] != ppdw[q])
            k = e[0][k]                                  //the e[k][0] works the same way as the π[k] in the
        if(ppwd[k+1] == ppwd[q])                         //textbook while the e[k][1] is how many words are in
            k = k + 1                                    //the longest prefix that is a suffix
            if((ppwd[k+1] == " " and ppwd[k+2] != None) or (ppwd[k+1]) == " " and ppwd[k+2] != " ")):
               spaces = spaces + 1
               //counts the number of spaces in the longest prefix that is also a suffix
        e[q][0] = k
        e[q][1] = spaces
        spaces = 0
   return e
```

pseudo for LCSs

```
max_similarity = 0                                       //measures maximum similarity between documents
                                                         //using discussed measurement (I am going to explain
                                                         //this in the note bit)
for i in range(length(corpus)):
   j = 0
   i = 0
   lcs = LCS(ppwd, corpus[i])
   if(lcs/min(corpus[i], ppwd)  > max_similarity):
        max_similarity = lcs/min(corpus[i], ppwd)  > max_similarity
max_similarity = max_similarity *100                              //convert from decimal to percent
if(max_similarity > 24):
    print("this document shows signs of potentially being plagiarized and has a lcs that contains" max_similarity "%
            similarity with at least one tested document")
else:
    print("this document has similarities that are within tolerance and has not likely been plagiarized")

LCS(ppwd, corp)
    lcs = [[0 for i in range length(corp)] for j in range(ppwd)]
    running == True
    while(running):
        if(i == ppwd.length-1 or j == corp.length-1):
            running == false
        elseif(ppwd[i] == corp[j]):
            lcs[i, j] = 1 + LCS[i+1, j+1]
        else:
            lcs[i,j] = max(lcs[i-1, j-1], lcs[i, j-1])
    return lcs[length(corp)-1, length(ppwd)]
```

Algorithm Analysis. Include the analysis of the algorithm and proof of correctness and running time of your algorithm.

Proof of correctness for kml:

Proof by loop invarience:

for this method to be correct the count and tc (temporary count) must accurately display the number of words in groups of 3 or greater and words that could potentially be in groups of 3 or greater respectively that are present in both the document in question and the corpus of documents. At the end of the loops all the tc = 0 and count must = all matched strings of length > = 3. Also, without loss of generality I will do a proof that is

- Initialization: start of a loop
  - Before a loop has run no letters have been checked and so the count is correctly placed at zero and the temporary count is also correctly placed at 0
- maintenance: if it is true at the start is it true later
  - Case 1: character between copus document and document being checked is the same
    - In both cases, i is set to i + 1 at the end so that the next loop is ready to go and we are not comparing the same element repetitively.
    - Case 1.1: End of word:
      - then tc = tc + 1 and count is unchanged, count is then correct by assumption and tc is also correct as a new potentially in a group of 3 matched wor has been added to the temporary count and since the temporary count was correct before this word was added by assumption, it must also be correct not
    - Case 1.2 Middle of word
      - then by assumption, tc and count are both correct already and we are not at the end of a word so no word can be added to either the count or tc which means these values remain unchanged which is correct.
  - Case 2: the characters are not the same
    - 2.1: q > 0
      - If the characters being compared are not the same then we must look at the e matrix. e[0][q] where q = the index of the character in the algorithm that is being compared with the qth character in the corpus document, e[0] = matrix storing longest prefix that is also a suffix for the qth character in the document being checked for plagiarism, and e[1] which simply stores the number of spaces in the current longest prefix that is also a suffix in the document that is being looked at. First, a temporary variable is set to q then

the qth character is set to the e[0][q]th character. This character is the best character that can be used to compare the least amount of characters and the proof for the e[0] matrix working can be found in Cormen, Thomas H. et al. *Introduction to Algorithms*, MIT Press, 2009. pages 1007 - 1008 (it is presented as the π matrix). After q is set to the e[0][q]th character, to stop words from being counted multiple times, e[1][temp] is accessed which records the number a spaces in the longest prefix that is also a suffix, this number is then placed in the variable prefix_spaces. Prefix spaces is then temporarily subtracted from tc to prevent words from being counted 2+ times. Then the size of tc is checked and if tc > 3 then count = tc which then makes the count accurate and then tc is set to tc = 0 to remove the words that have been added to the count and then tc = prefix_spaces which then adds back the words that are now potentially words in groups of greater than 3, this then makes it so that tc is also accurate. Thus maintenance is maintained.

Case 2.2: q = 0

If q = 0 then it is a nice simple case, q !>0 so the while loop breaks, the if statement for just this eventuality it run, if tc > 3, then count = tc + count, and tc = 0. And because both tc and count are correct prior to the run, they must also be correct after as they have been correctly added. Then the ith index in the corpus document is moved up by 1 so that we are ready for the next loop

Thus maintenance holds in all cases

Termination:

Case 1 End of file -- character between copus document and document being checked is the same:

The final for loop is broken and the tc is checked to see if it is greater than 3, if it is then count = count + tc, and because count and tc are by assumption correct before the final/terminating loop, the new count must also be correct. Then tc is set to zero which gives exactly what is desired.

Case 2 End of file -- character between corpus document and the document being checked are different:

in this case q will = 0 and so the while loop will be broken (if started) and case 2.2 will take over, but at the end i = length of corpus document and so the for loop is broken and the count and tc are accurate for the same reason as in cases 2.2. The final for loop will not affect values as tc = 0 and thus the termination gives the desired result.

Thus, as initialization, maintenance, and termination are valid in this algorithm the algorithm must be correct. This particular proof was done with a corpus of one document; however, but only difference is that the algorithm runs through multiple documents counting all of the words in groups of 3+ in all if the document in the corpus and the document being checked for plagiarism. Thus this algorithm is correct (Note: this proof leans on the correctness of the π function proof from Cormen, Thomas H. et al. *Introduction to Algorithms*, MIT Press, 2009. pages 1007 - 1008 which is an additional 2 pages)

Proof of correctness for LCSS:
Induction can be used to prove the correctness of LCSS.

Base Case: The first step of the algorithm is the first iteration of the loop, where the first character of m and n are compared. So the problem becomes finding the longest common substring between two single character strings, for example, “a” and “b”. It is obvious if the characters are the same then the character itself would be the longest common substring

Inductive Step: Since step 1 is proved to be true, then step i of the algorithm would also be true. In that case at step i+1 the next character of the strings will be compared

Case 1: If the characters are the same they would add to the string, they would be added to the substring at step i and be saved. Since at step i the algorithm already produces the correct longest common substring, than the added character would obviously create the current longest common substring.

Case 2:If the characters are different, then the characters at step i+1 are no longer common, so the longest common substring would be from step i , which is proven to be correct, would be saved

Because at step 1 the algorithm is proved correct, and so step i is also correct, then by inductioni+1 will also be correct, and the algorithm would produce the correct result all the way until the end of the array, where the element stored would be the longest common substring, and the algorithm would terminate

Time Complexity/Running time for KMP
The KMP algorithm has a time complexity of O(n+m), where n is the length of the text and m is the length of the pattern. For each document in the corpus, the KMP algorithm is called once, so the time complexity of the PlagiarismDetector function is O(n*(n+m)). So the time complexity using the notation would be O(n * di + P)

running time LCSS (Time complexity is O(mn) I believe rather than O(m + n) in the textbook, I will have to look at a bit)
The first step of the algorithm is to create a table or a 2d array of size m*n, where m and n are the size of the two strings to be compared. The initial values are all zero and take an approximate constant time of c, so this means the total time complexity would be

$$c * m * n = O(mn)$$

Then the algorithm will loop through each index of the array and compare the corresponding character at the same index in string m and n. The two characters are compared in a constant time c_1, if they are the same they will be added to the biggest neighbor and save in this index of the array. If the characters are different, then only the biggest neighbor is added. Due to the way, the array is iterated there will be only three neighbors, and the time to search for a single neighbor is also a constant time c_2. These steps are repeated for the entire array which has the size of m*n, So in total the time complexity is

$$m * n * (c_1 * 3 * c_2) = O(mn)$$

Then only the last index of the array would store the longest common substring. To access it would be a simply constant time $c_1 = O(1)$, so the total time complexity of the whole algorithm is

$$O(mn) + O(mn) + O(1)$$

Taking the most significant growth rate results in

$$O(mn)$$

Unexpected Cases/Difficulties: If you encountered any issues for design and implementation of your work, include it here. Also, mention the solutions you have considered to alleviate the issue.

General: It took some time the think of how to implement both methods to check over more than one document as both methods were designed to compare one document at a time to fix this problem we, to fix the issue with capitalization and punctuation differences we converted all the used text to lowercase and we also removed all punctuation

LCS: Because LCS does not look for specific strings this presents a problem for what we defined to be plagiarism, to remedy this particular problem we will redefine plagiarism for this method so a plagiarized document will be one that has the longest common subsequence that is equal to the largest value obtained from taking the max{lcs[i]/(min{length(ppwd), length(corpus[i])})} (∀i ∈ ℕ ∈ corpus.length) where lcs[i] is defined as the longest common subsequence between the document being checked for plagiarism and the i^th document in the corpus and min{length(ppwd), length(corpus[i]} = the length of the shorter of the 2 documents, this method was taken from https://reader.elsevier.com which is very interesting document. This presents an issue if a long document is compared with a short document which is addressed in the document mentioned; however, do to time constraints we will not be impementing the fix. The fix used is using an additional measure of local lcs

KMP: One potential issue with the proposed algorithm is that it might produce false positives. For example, if a common phrase or sentence is present in both the corpus of documents and the potentially plagiarized document, it might be identified as plagiarism even if it is not. We can use stop-word removal or stemming to reduce the impact of common phrases.
Another issue is that the time complexity of the algorithm can be quite high, especially for large set of documents. To address this issue, we can use techniques such as indexing or hashing to speed up the matching process.
Another potential issue is that the algorithm might not detect plagiarism if the copied text is heavily modified or paraphrased. To address this issue, we can use techniques such as cosine similarity or latent semantic analysis (LSA) to compare the semantic similarity between the documents, instead of relying only on exact word matches

Task Separation and Responsibilities. Who did what for this milestone? Explicitly mention the name of group members and their responsibilities.

Samuel S: Pseudocode for (LCSS, KMP), part of unexpected cases, Proof of Correctness for KMP

Sam G:: Problem Formulation for both algorithms, part of unexpected cases and difficulties (KMP), Running time and Time Complexity (KMP)

Jimmy J.: Time Complexity and Proof of Correctness (LCSS)