Coursework1

dan.willis.uni

November 2019

1 PartA.1

Algorithm 1 calculateFeatureVector(**A**[],**Q**[]) **return** F

Require: a Dictionary **Q**
Require: a Document **A**
Require: an Empty array **F**
Require: Tree **a**

for $i \leftarrow 1$ to w **do** ▷ *for every word*
    a.add(**A**[**i**]) ▷ *add to tree*
for $i \leftarrow 1$ to s **do** ▷ *for every word in dictionary*
    $F[i] \leftarrow a$.countNode(**Q**[**i**]) ▷ *count*
return F

Algorithm 2 add(**str**)

Require: *str*

root ← addrecursive(*root*,*str*) ▷ *Goes into recursive to add to the correct place*

2 PartA.2

This is the stripped down psuedo code for the algorithm This means that the algorithem has a run time complexity of w(addRecursive) + s(countNode) in the best, worst and average case. It fully depends upon the other algorithems as to how quick it is. There are n nodes in the tree
The hmax the maximum height of the tree is n where each node only has one child
The havg is the average height of the tree which is log(n)
The hmin is the minimum height of the tree which is 1 where the root only has one child so the other child is free unless the tree is empty in which case it is 0
addRecursive Time complexity:

Algorithm 3 addRecursive(**currentNode**,**str**)**return** *node*

Require: current Node infomation $currentNode$
Require: String to search for str
if $currentNode = null$ **then** ▷ *if the node being looked at is null* **return** new $Node$ ▷ *Return the null node to be populated*
if $currentNode > str$ **then** ▷ *if current node greater than the search string*
  $currentNodeLeftChild \leftarrow$ addRecursive($currentNodeLeftChild$,str) ▷ *search left*
if $currentNode = str$ **then** ▷ *if they are equal*
  $currentNodeLeftChild \leftarrow$addRecursive($currentNodeLeftChild$,$str$) ▷ *search left*
if $currentNode < str$ **then** ▷ *if node less than string*
  $currentNodeRightChild \leftarrow$ addRecursive($currentNodeRightChild$,str) ▷ *search right*
return $currentNode$

Algorithm 4 countNode(**str**)**return** *num*

Require: string to search for str
Require: $num \leftarrow 0$ ▷ *set the number to 0*
countNodeRecursive($root$,str) ▷ *count recursive* **return** *num*

Algorithm 5 countNodeRecursive(**currentNode**,**str**)**return** *num*

Require: *num*
Require: Current Node infomation **currentNode**
Require: String to search for **str**
if $currentNode! = null$ **then** ▷ *if current node not null*
  if $currentNode < str$ **then** ▷ *if node less than string*
    countNodeRecursive($currentNodeRightChild$,str) ▷ *count right*
  else
    if $currentNode = str$ **then** ▷ *if eqaul*
      $num \leftarrow num + 1$ ▷ *add one to count*
    countNodeRecursive($currentNodeLeftChild$,str) ▷ *search left*

Algorithm 6 calculateFeatureVectorStripped(**A**[],**Q**[]) **return** F

for $i \leftarrow 1$ to w **do** ▷ *for every word*
  $root \leftarrow$ addrecursiveStripped($root$,$A[i]$) ▷ *Goes into recursive to add to the correct place*
for $i \leftarrow 1$ to s **do** ▷ *for every word in dictionary*
  $F[i] \leftarrow a$.countNodeStripped(**Q**[**i**]) ▷ *count*
return F

Algorithm 7 addRecursiveStripped(**currentNode**,**str**)**return** *node*

if $currentNode = null$ **then** ▷ *if current node null* **return** new $Node$ ▷ *return node to be populated*
$Child \leftarrow$ addRecursiveStripped($Child$,str) ▷ *child node = recursive of that child* **return** $currentNode$

Worst Case is O(hmax)
Average case is O(havg)
Best Case is O(hmin)
This is because it has to traverse the tree until it finds a null child and then add there
CountNode Time Complexity:

Algorithm 8 countNodeRecursiveStripped(**currentNode**,**str**)**return** *num*

if $currentNode! = null$ **then** ▷ *if node not null*
    countNodeRecursive($Child$,str) ▷ *count child*

Worst Case is O(hmax)
Average Case is O(havg)
Best Case is O(hmin)
Because it traverses the depth of the tree

Worst Case:

$$\sum_{n=1}^{w} n + \sum_{n=1}^{s} n$$

$= \frac{1}{2} w\,(w+1) + \frac{1}{2} s\,(s+1)$
O($w^2 + s^2$)
Average Case:

$$\sum_{n=1}^{w} log(n) + \sum_{n=1}^{s} log(n)$$

$= log(w!) + log(s!)$
$= wlog(w) + slog(s)$
O($wlog(w) + slog(s)$) Best Case:

$$\sum_{n=1}^{w} 1 + \sum_{n=1}^{s} 1$$

$= w + s$
O(w+s)

3 PartA.3

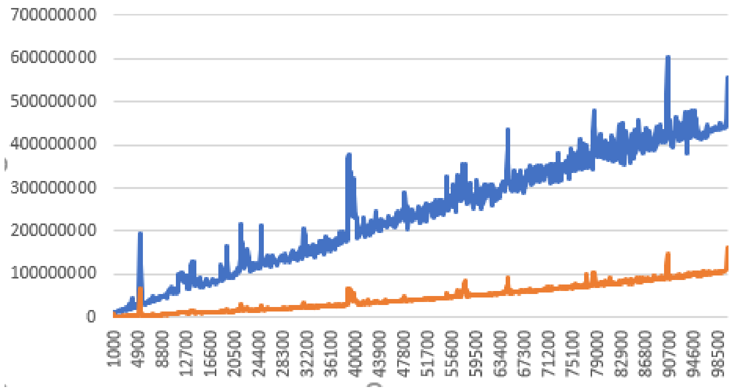
See Java Code

4 PartA.4

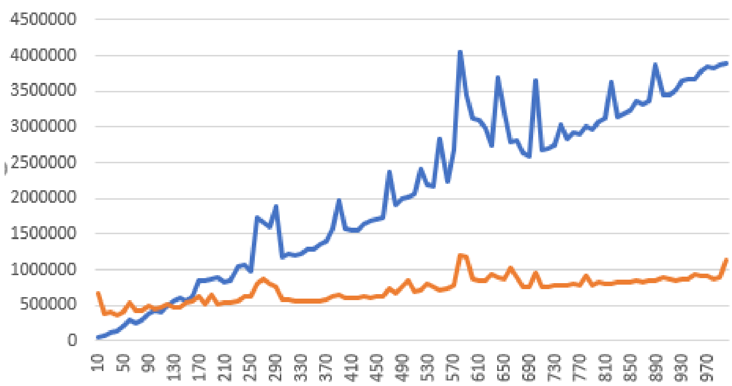
The way that I conducted all the tests was for every test I ran I did it five times and then averaged those five times. For each of the five times, a different set of inputs was created and fed into both the algorithms. Then the output was checked to be the same by both. I did all the times in nanoseconds as this was the most accurate I could make it. All the tests ran overnight at the same program one after another and all the results of all the tests had no difference between the calculations.

Figure 1: Feature Vector Test Results

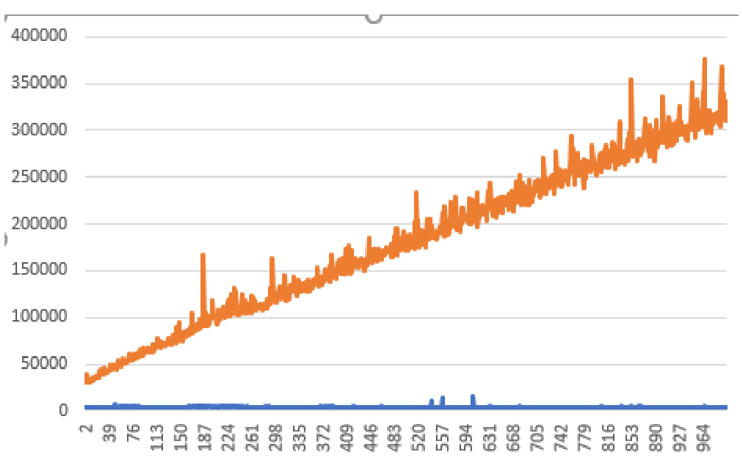
(a) Test One Results



(b) Test Two Results



(c) Test Three Results



1a is a graph of the first test I did which was varying the number of words in the document (w) from 1000 to 100000 incrementing by 1000 each time. This gives me 100 pieces of data. The lower line is the algorithm I used in the end and the other line represents the brute force algorithm. As you can see the gradient of the line representing my algorithm is much less than the gradient of the other line. This means that it is far quicker for a sufficiently large w. The other values were that the dictionary had 1000 words and all the words were 5 letters long.

1b is the results of test two in which I varied the size of the dictionary (s) from 10 to 1000 incrementing by 10 each time again giving me 100 pieces of data. Initially, the brute force algorithm is faster than mine for the lower dictionary size and is faster up until a dictionary size of 130 words. While my algorithm starts slightly higher its gradient is far less than the brute force algorithm so that begins to take longer. This means that for a sufficiently large s it is faster. The other values were 1000 words in the document all of which are 5 characters long.

1c is the results of test three in which I varied the length of the words in the dictionary and the document. From 2 to 1000 incrementing by 1 each time. For this test brute force was quicker every time as the time taken to complete the task doesn't depend upon the length of the words. Whereas my algorithm does because it sorts the words into order and the longer they are the longer it takes to sort them. Therefore my algorithm took longer and had a higher gradient so for a sufficient word length the brute force algorithm would be quicker. The other values were a dictionary size of 10 and a document length of 100.

Test four doesn't have a graph as there is only one piece of data per algorithm. This was just testing the difference in time when the dictionary size and document size were both set high and the word length was kept at 5. The dictionary size was 10000 and the document size was 1000000 this took the brute force algorithm on average 213381 ms. And took my algorithm 3159.700740 ms which is 67.5 times faster.

5 PartB.5

Algorithm 9 calculateDSD(**A**[], **B**[])**return** *DSD*

Require: DSD
Require: Feature Vector A **A**[]
Require: Feature Vector B **B**[]

$DSD \leftarrow 0$
for $i \leftarrow 1$ to s **do** ▷ *for each dictionary word*
    $DSD \leftarrow DSD + —A[i] - B[i]—$ ▷ *add absolute of A-B*
return DSD

---

6 PartB.6

Algorithm 10 findNearestDocument(**Docs**[][], **Q**[])**return** $FND[]$

```
Require: An Array of Documents Docs
Require: A dictionary Q
Require: A Empty Array FND
Require: Number Of Doucment n
Require: A Distance array Distance[n][n]
Require: Feature Vector Array F[n][]
for A ← to n do                                          ▷ for Each Document
    F[A] ← calculateFeatureVector(Docs[A],Q)             ▷ calculate the feature vector
    distance[A][A] ←high                                 ▷ set distance to itself high
for A ← 1 to n do                                        ▷ for each document
    currentBestDistance ←high                            ▷ set closest document to NULL
    currentBestIndex ← 0                                 ▷ set closest documenet to none
    for A2 ← 1 to n do                                   ▷ for each document
        if distance[A][A2] is null then                  ▷ if the distance is null
            distance[A][A2] ←calculateDSD(F[A],F[A2])    ▷ calculate DSD
            distance[A2][A] ← distance[A][A2]            ▷ set mirror
        if distance[A][A2] ¡ currentBestDistance then    ▷ if the document is closer than the closest so far
            currentBestDistance ← distance[A][A2]        ▷ set the current best distance
            currentBestIndex ← A2                        ▷ se the current best index
    FND[A] ← currentBestIndex                            ▷ set the return array
return FND
```

7 PartB.7

This is the stripped pseudo code

Time Complexity $= n(\text{calculateFeatureVector}) + \frac{n(n-1)}{2}(\text{calculateDSD})$

Algorithm 11 findNearestDocumentStripped(**Docs**[][], **Q**[])**return** $FND[]$

```
for A ← to n do                                          ▷ for each document
    F[A] ← calculateFeatureVector(Docs[A],Q)             ▷ calculate F
for A ← to n do                                          ▷ for each document
    for A2 ← to n do                                     ▷ for each document
        if distance[A][A2] is null then                  ▷ if uncalculated
            distance[A][A2] ←calculateDSDStripped(F[A],F[A2])    ▷ calculate DSD
return FND
```

Time complexity analysis of calculate DSD All Best, Worst and Average Cases

Algorithm 12 calculateDSDStripped(**A**[], **B**[])**return** DSD

for $i \leftarrow 1$ to w **do**
    $DSD \leftarrow DSD + —A[i] - B[i]—$
return DSD

are the same which is O(s)
Therefore
WorstCase:
$n(\frac{1}{2}w\,(w+1) + \frac{1}{2}s\,(s+1)) + s(\frac{n(n-1)}{2})$
$= \frac{ns^2+n^2s+w^2n+wn}{2}$
$=\mathrm{O}(ns^2 + n^2s + w^2n)$

Average Case:
$n(wlog(w) + slog(s)) + s(\frac{n(n-1)}{2})$
$\mathrm{O}(n^2s)$

Best Case:
$n(w+s) + s(\frac{n(n-1)}{2})$
$= nw + \frac{n^2s+ns}{2}$
$\mathrm{O}(n^2s)$

8 PartB.8

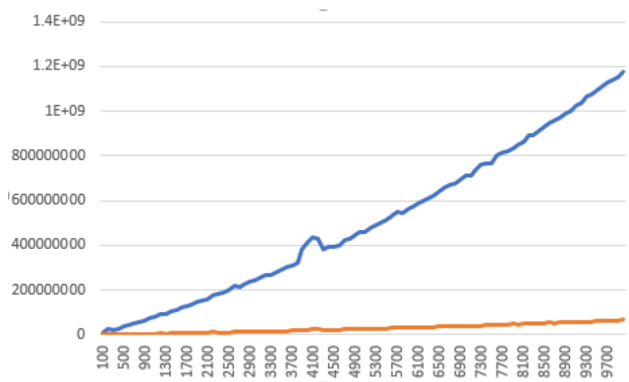
See Java Code

9 PartB.9

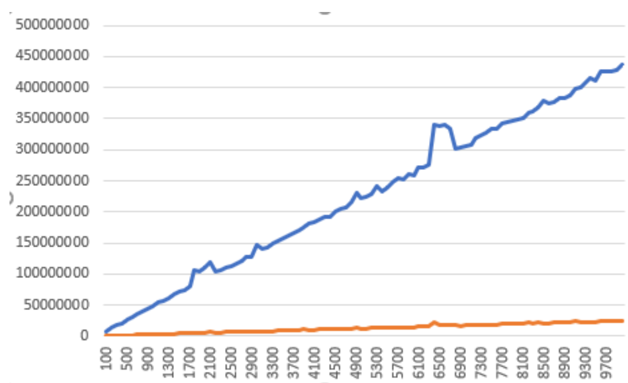
Both of the document similarity functions use the same algorithm to calculate the feature vectors so that it is just the algorithm that makes a difference.

Figure 2: Document Similarity Test Results

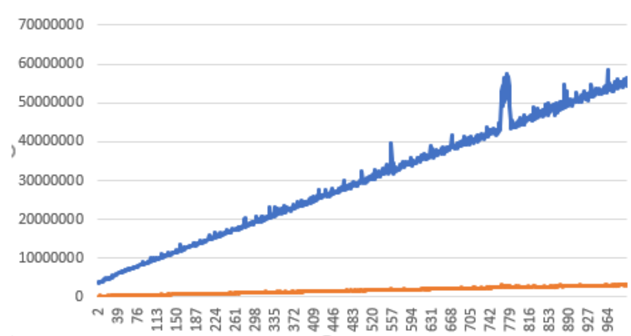
(a) Test Five Results



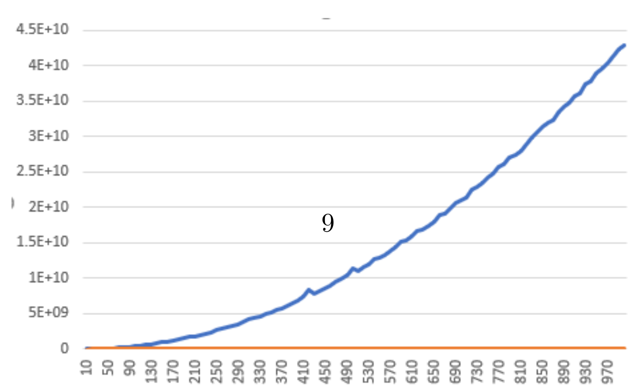
(b) Test Six Results



(c) Test Seven Results



(d) Test Eight Results



2a is the results of test fives, where I changed the document size from 100 to 10000 incrementing by100 each time. Other values where a dictionary size of 100, a word length of 5 and 10 documents. My algorithm is the lower line and has a much lower gradient than the brute force algorithm. This means that it is much faster and will be for a sufficiently large w.

2b is the results of test six, where is changed the dictionary size from 100 to10000 incrementing by 100 each time. The other values were a document size of 100, a word length of 5 and the number of documents being 10. The lower line is my algorithm and has a lower gradient meaning for a sufficiently large s it is still faster.

2c is test seven, where I changed the length of each word from 2 to 1000. The other values where 100 words in a document, 10 words in a dictionary and 10 documents. The gradient of the brute force algorithm is significantly more than my algorithm. So, for a significantly large length of words, my algorithm will be faster.

2d is test eight where I changed the number of documents from 10 to 1000 incrementing by 10 each time. The other values are a document length of 100, a dictionary length of 10 and a word length of 5. The gradient of the brute force has a definite curve of a quadratic which makes sense whereas mine looks relatively linear. And is significantly lower gradient meaning that for a significantly large number of documents my algorithm is faster.

Test nine is just two pieces of data with 100 documents, a dictionary size of 10, word length of 5 and the number of documents being 10000. This took the brute force algorithm on average 4297.7161401 seconds and took my algorithm 4.8981888 seconds on average. This is 877.4 times faster.