Université Jean Monnet Saint-Etienne
Faculté des sciences et techniques

Project on Text Formatting Problem

Aleksei Pashinin,
Aninda Maulik,
Dhayananth Dharmalingam,
Poulomi Nandy,
Rediet Tadesse

December 2019

1 Introduction

The objective of this project is to implement algorithms for solving text formatting problems or printing neatly,there are three types of alignment left,right and center.To provide an experimental study of their running time and the quality of the solutions found.

2 Description

In order to provide solution for this particular problem of text formatting,we have provided below mentioned(6) approaches,and we also have the experimental analysis of the respective problem.

Algorithm 1

Our 1st approach is ***Dynamic Approach***:

The deficiency of first idea lies in that it repeatedly solves the same sub problems. Yet suppose there was an optimal configuration of lines. Plucking off its last line would still keep the layout optimal because otherwise it would be possible to improve it and, together with the removed line, would result in even better configuration, contradicting its optimally. To solve each sub problem just once, it is then necessary to find out and later reuse which of the lines ending with some word contributes least to the overall cost.Dynamic Programming is a solution for solving a hard problem by separate it into a number of simpler sub-problems, solving each of this simple sub-problems just one time, and stock this solutions in memory-based data structures. Each of the solutions is indexed with order which based on the values of its input parameters. When the next time the same sub-problem will be, we can simply looks on the array of previously computed solutions, and it will economize the computation time.

PsuedoCode for Dynamic Approach:

```
    FUNCTION Dynamic(text, width):

    words <-text.split()
    wordsize <-len(words)
    totalSpaces<-[][]
    result<-[]
for i from 0 to sizeL:
totalSpaces[i][i] <- width-len(words[i])
```

```
for j from i + 1 to sizeL:
totalSpaces [i][j]<-totalSpaces[i][j-1]len(words[j])-1
    ENDFOR
   ENDFOR
    for i from 0 to sizeL :
        for j from i to sizeL:
        IF totalSpaces[i][i]<0 :
            totalSpaces[i][i] <- MAX.VALUE
ELSE:
            Math.pow(totalSpaces[i][j], 2)
ENDIF
       ENDFOR
ENDFOR
    for i from sizeL-1 to 0:
        minCost[i]<-totalSpaces[i][sizeL-1]
        result[i]<-sizeL
     for j from sizeL-1 to i:
        IF totalSpaces[i][j-1]=MAX_VALUE):
        CONTINUE
    IF minCost[i] > minCost [j]+totalSpaces[i][j-1]:
    minCost[i] <- minCost[j] + totalSpaces[i][j-1]
    result[i] <- j
        ENDIF
       ENDFOR
ENDFOR
    OUTPUT result
ENDFUNCTION
```

Algorithm 2

Our 2nd approach is ***Greedy Approach***:

The greedy solution is to place as many words as possible in the first line. Then do the same thing for the second line and so on until all words are placed. This solution gives optimal solution for many cases, but doesn't give optimal solution in all cases. For example, consider the following string "Cat is on table " and line width as 6. Greedy method will produce following output.
Cat is
on
table
Extra spaces in the above 3 lines are 0, 4 and 1 respectively. So total cost is $0 + 4^3 + 1 = 65$. In this case it can also be written as:
Cat
is on
table
So,the extra spaces could have been are 3,1 and 1 respectively. So, the total cost is $3^3+1+1=29$. Hence,it is clear that Greedy approach may not be the optimal one.

PsuedoCode for Greedy Approach:

```
Greedy_Print(words)
    m=10;
    string solution;
    cost=0;
    line=0;
    for(word.count)
      {
        space_used = space_used+word(i).length();
        if(space_used<=m)
```

```
        {
            if(i==0)
             {
                solution=word(i)
             }
            else
             {
                solution=solution+" "+word(i)
             }
            lc=m-space_used;
            space_used++;
        }
      else
        {
            linecost[line]=lc*lc*lc;
            solution=solution+"/n";
            lc=0;
            space_used=0;
            line++;
        }
    }
```

Algorithm 3

Our 3rd approach is ***Space Optimization Approach***:

Space Optimized Solution is based on idea to use two 1-D arrays, where the value i of the first array represents minimum cost of the line in which arr[i] is the first word and the value i of the second array represents index of last word present in line in which word arr[i] is the

first word. Suppose for any line the first word is value in arr[i], and last word is value in arr[j].The minimum cost of that line is stored in i of first array. We will check over all values of j and do attention of number of characters added so far in line. Finally we need to find cost of current line, and compare this cost with minimum cost to change value of minimum cost (first array) and update if we need it. We will repeat above procedure for each value of i from 1 to total number of words, to get best solution.

PsuedoCode for Space Optimization Approach:

```
FUNCTION SpaceOptimized(text, width):

words <- text.split()
wordsize <- len(words)
minCostLine<-[]
result<-[]

minCostLine[sizeL - 1] <- 0;
result[sizeL - 1] <- sizeL-1

for j from sizeL - 2 to 0:
        nbOfCaracters = -1
        minCostLine[i] = MAX_VALUE
        for j from i  to sizeL:
    nbOfCaracters <- +(ListOfWords[j+1]+1)
    IF nbOfCaracters > SizeLine:
        BREAK
    ENDIF
```

```
ENDFOR
IF j == sizeL1 :
        minCost<- 0
ELSE
minCost<-(SizeLine-nbOfChar)*
         (SizeLine - nbOfChar)+minCostLine[j + 1]
       ENDIF
       IF minCost < minCostLine[i] :
                    minCostLine[i] = minCost
                    result[i] = j
           ENDIF
           ENDFOR
     ENDFOR
     OUTPUT result
ENDFUNCTION
```

Algorithm 4

Our 4th approach is ***Brute Force Approach***:

The main idea of brute force is dividing a sentence into a sequence of all possible combination of consecutive words and put them in in a line of fixed width and each time calculating cost and return the optimal one. It doesn't store any value(this makes it different from dynamic programming). How it works:

1.Given input sentence and screen display size of width,

2. Split sentence in to words

3. Try to put words in a line with every possible combination of words by considering width and each time calculate triple of cost of each line and sum up to get the total cost.

4. Update minimum cost every time by comparing with the previous.
5. Return the optimal solution which have minimum total cost.

E.g Sentence:-The dog is alive=7, Min cost=infinity

Table 1: Combination 1.1:putting every word in a single line

Words#1	Cost of Line#2
The	7-3=4*4*4
dog	7-3=4*4*4
is	7-2=5*5*5
alive	7-5=2*2*2

Total cost=64+64+125+8=264
Min cost=264

Table 2: Combination 1.2:

Words#1	Cost of Line#2
The dog	7-6=1*1*1
is	7-2=5*5*5
alive	7-5=2*2*2

Total cost=1+125+8=134
Min cost=134

Table 3: Combination 1.3(we can't consider this combination since it is negative):

Words#1	Cost of Line#2
The dog is alive	7-10=(-3)*(-3)*(-3)

Note that we didn't consider all the combinations: The result will be The one with min cost which is:
The

Dog is
alive

PsuedoCode for Brute Force Approach:

```
from itertools import combinations, chain
FUNCTION powerset(iterable):
    s <- list(iterable)
    RETURN
    chain.from_iterable(combinations(s, r)
    for r in range(len(s)+1))
ENDFUNCTION
 ENDFOR
FUNCTION bruteforce(text, width):
          ENDFOR
    words<- text.split()
    count <- len(words)
    minimum <- float("inf")
    breaks <- ()
    for b in powerset(range(1, count)):
     m <- 0
       i <- 0
          for j in chain(b, (count,)):
              w <- len(' '.join(words[i:j]))
              IF w > width:
                   break
              ENDIF
              m += (width - w) ** 2
              i <- j
          ELSE:
```

```
                IF m < minimum:
                    minimum <- m
                    breaks <- b
                ENDIF
    ENDFOR
        ENDFOR
    lines <- []
    i <- 0
    for j in chain(breaks, (count,)):
        lines.append(' '.join(words[i:j]))
        i <- j
    ENDFOR
    OUTPUT lines
    RETURN lines
ENDFUNCTION
```

Note that we didn't consider the repeated combinations The result will be The one with min cost which is: The Dog is
alive

We have to note that Brute Force will not work for large number of words.

Algorithm 5

Our 5th approach is ***Branch and Bound Approach***:

In this approach we will check all the solution and we would explore the branch which gives the least cost and bound all the irrelevant branches.Only the tree with least cost or count will be branched forward.For example:
"Cat is on table"the given sentence has w(line width)=6.Branch

and bound will produce following output:
Cat is
on
table
Total cost= $0+4^3+1^3=65$
Here, in this approach it is taking one word and subtracting every time from w in this case,it has taken 'cat' and 1st branch(6-3=3),2nd branch is by taking the next word which is 'is' branch 2 including space is (6-6=0),it will stop here as it got the postive minimum value and will explore the second branch from here on. So, from the above example we got that the cost is 0 in the first line.It will collect the cost of each line and cube the number.Finally it adds all the cost we got from each line.This also clearly states that not always Branch and bound approach will give us optimal solution

PsuedoCode for Branch and Bound Approach:

```
CostArray=[line]
 m=10
line_and_words=[line],[i];
cost=[line];
wordlist=[]
wordlengthlist=[];
mincost=m=10;
for(words)
      CurrentCost=m-word[i].length()+CurrentCost
      if(CurrentCost<= 0)
          CostArray[1]=mincost
          1++
```

```
        CurrentCost=0
        total_cost+=mincost
        word=0
        CurrentCost=0
        line_and_word[line][word]=wordlist[i];
    else
        line_and_words[l][i]= word[i];
        if(mincost>CurrentCost)
            mincost=CurrentCost;
    CurrentCost= m-wordlist.get(l).lenth();
      if(mincost > CurrentCost)
        mincost= CurrentCost
```

Algorithm 6

Our 6th approach is ***Divide and Conquer Approach***:

Given,text containing k characters for which we would have to compute the number of characters which can fit in the line, whose number of characters:-L is defined by the user.

Compute k/L which would give a quotient and a remainder. The quotient value is the number of lines that's required to print the given text. The remainder value would generate the final line, which is one additional to the above generated line due to the quotient value.

Moreover, if we have a fraction of a word at the end of the first line; then that fractional word needs to be removed. This removal of the fractional word is done by comparing the weight value of the indexed word to the weight of the input indexed word. Eg: If the 4th word is 'honey' with weight 5 and the weight of the word at

the end of the first line is 'ho' with weight 2 then this 'ho' needs to go off from the first line. The 'ho' would be now added to the 2nd line; and likewise we need to find out if the last word of the 2nd line has similar impacts on the corresponding next line, or not. If it does then we perform the above procedure and if the impact doesn't show up there then the program would simply move to the next line. We have to repeat similar steps till the last line.

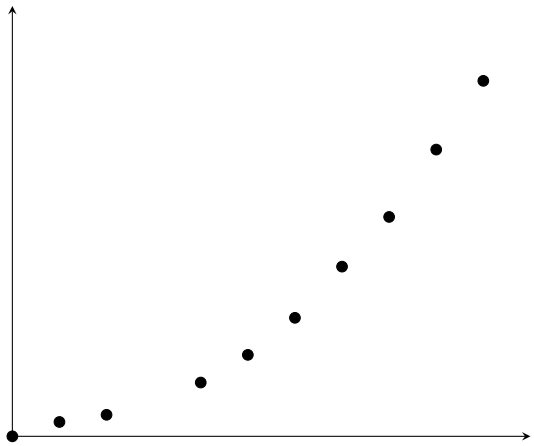
PsuedoCode for Self Approach:

```
split_and_sort((text,wordArray)
 m=10
 lc=0
 TotalCost=()
 lineCount= text.length()/m;
 if(remember>0)
  linecount++
 List<string>divided string= new List()
 for(int 1; 1<= lineCount;1++)
     divided.add(wordText.substring(S,M);
     s+=m;
 for(divided in Item)
  lastword_index= item.split("\\s").count
  lastword= item[lastword_index-1]
  if(lastword.length!= Actualword.length)
      divided.add(lineCount+1,lastword)
      divided.remove(lineCount,lastword)
  if(divided[lineCount].length()>m)
 extra= divided[lineCount]/m
```

```
divided.add[lineCount++,extra]
divided.remove[lineCount,extra]
lc=divided[lineCount].split("\\S+").count
TotalCost+=lc*lc*lc
```

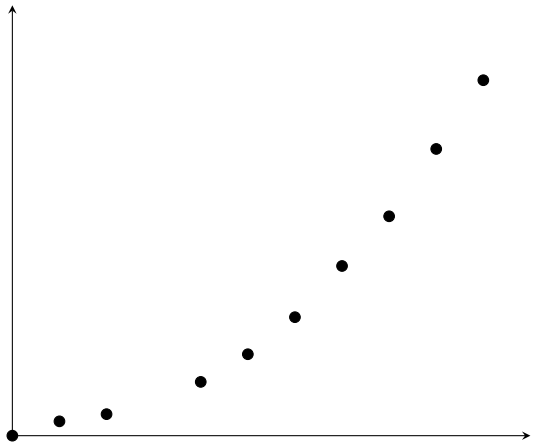
3 Experimental Studies

Plot for Dynamic Approach:



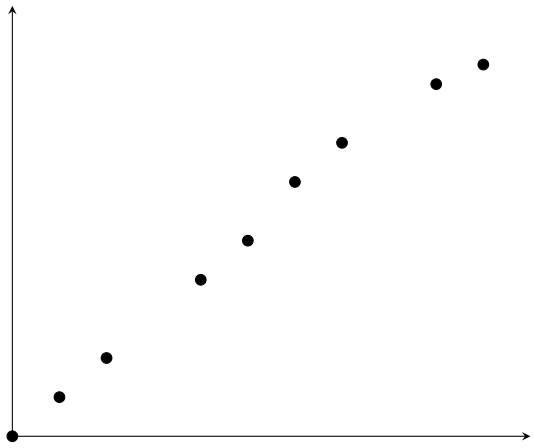
X-axis denotes no.of word
Y-axis denotes time

Plot for Space Approach:



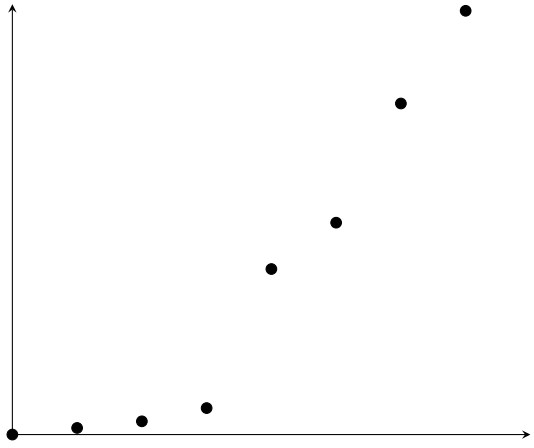
X-axis denotes no.of word
Y-axis denotes time

Plot for Greedy Approach:



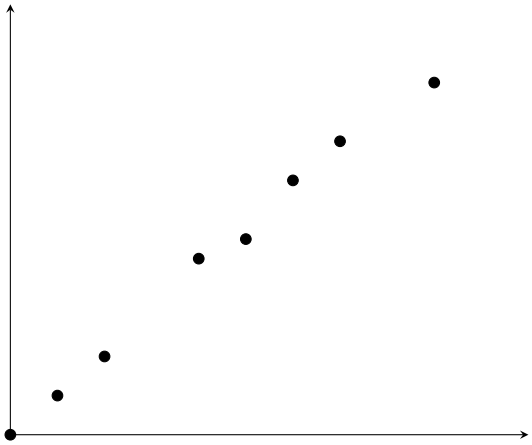
X-axis denotes no.of word
Y-axis denotes time

Plot for Brute and Force Approach:



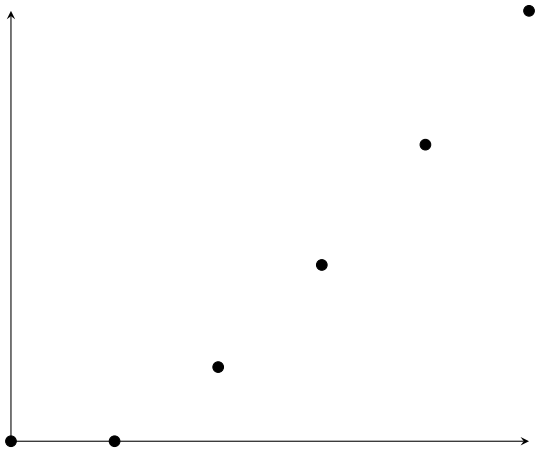
X-axis denotes no.of word
Y-axis denotes time

Plot for Branch and Bound and Force Approach:



X-axis denotes no.of word
Y-axis denotes time

Plot for Divide and Conquer and Force Approach:



X-axis denotes no.of word
Y-axis denotes time

4 Results

After doing a close analysis, we can conclude that Dynamic programming gives the best result in terms of

getting the minimum cost as compared to other algorithms that have been included in our report such as Greedy,Branch and bound, Brute force, Self Optimisation,Self Method. However, in accordance to our codes, we found out that the time complexities are a follows. Brute force:-O(2^n) Dynamic programming method:-O(n^2) Greedy method:-O(n) Branch and bound:-O(n)

These two methods are our proposed method and below are the time complexities. Divide and Conquer Method:-O(nlogn) Space optimization:-O(n^2)

5 Work Distribution among the team

Aleksei Pashinin developed the GUI and wrote an algorithm on Dynamic approach and space optimization with the source code in Java language and compiled the same.Aleksei wrote his part of content with experimental analysis in the report

Aninda Maulik wrote the Divide and Conquer approach algorithm and coded the same in Java language and compiled the same.Aninda also helped in debugging the error coming up while running the code.Aninda wrote his part of content with experimental analysis in the report

Dhayananth Dharmalingam wrote the Greedy approach algorithm and coded the same in Java language and compiled the same.Dayananth also helped in debugging the error coming up while running the code.Dayananth wrote his part of content with experimental analysis in the report

Poulomi Nandy wrote the Branch and Bound approach algorithm and coded the same in Java language and com-

piled the same.Poulomi wrote her part of content with experimental analysis in the report and also merged all in one file

Rediet Tadesse wrote the Brute Force approach algorithm and coded the same in Java language and compiled the same.Rediet wrote her part of content with experimental analysis in the report