Word Wrapping Problem

Mehdi Lotfipour

All possible solution for word wrapping problem is the same as number of partitioning an array with the same sort and partitioning problem is in exponential family. But there is a better algorithm with dynamic programming method. Firsto we define a n by n matrix which n is total number of words in our sequence and we call it "Line Cost Matrix". Each element $LineCostMatrix_{i,j}$ is cost of putting i'th word to j'th word in one line. The formula for cost is given by:

$$LineCostMatrix_{i,j} = \left[M - (j-i) - \sum_{k=i}^{j} |word_k|\right]^3$$

If this cost is negative, we set it to positive infinity to discard it as a solution. If this value is positive but j equals n, it means we are in last line and it doesn't have any cost, so we set it to zero and in other cases we set it to the number we get.

Now we define a variable C_i that is minimum total cost if we use word from 1 to i. For minimum we must check different $C_k + LineCostMatrix_{k+1,i}$ values that $k = 1, 2, \ldots, i$. For gettnig C_i we need values of $C_1, C_2, \ldots, C_{i-1}$. By use of dynammic programming we can build array C from left to right. The index that make the minimum cost is kept in another array $BreakPoints$ for showing the result. We define a function which receives all words and this array to build the paragraph. This function start from last index and the value of it is the index of last line starter. Then we go for second line from bottom and ending word is in index of previous starter and index of second line starter is the value of new ending index.

In [ ]:
```
#importing libraries
import numpy as np
#defining useful functions
#this function calculate the cost if we put words
#from starting index to ending index in one line
def line_cost_cal(word_length_list, start_index, end_index, M):
    #inputs
    #word_length_list: a list containing length of words
        #first element is zero to reserve zero index and start from one
    #start_index: index of starting word
    #end_index: index of ending word
    #M: maximum length of line
    #output
    #cost of putting words from starting index to ending index in one line

    #helpful variables
    #n: index of last word
    n = len(word_length_list) - 1

    #getting cost
    cost = (M - (end_index - start_index) - np.sum(word_length_list[start_index: end_index + 1]))
    #if cost is negative, we return +infinity cause its impossible
    if cost < 0:
        return 100000
    #if its positive but:
    else:
        #if ending index is last index we return zero cause last line doesnt matter
        if end_index == n:
            return 0
        #if none of those we will return calculated cost
        else:
            return cost
```

In [ ]:
```
#this function print the words in some lines based on break points
def seq_printer(word_list, break_points):
    #inputs
    #word_list: a list containing all words
        #first element is blank string to reserve zero index and start from one
    #break_points: a list containing break points of sentence
    #output
    #printing the words based on break points

    #helpful variables
    start_index = len(break_points)

    #keeping each line in elements of list
    lines = []
    while True:
        #starting from last line
        end_index = start_index - 1
        start_index = break_points[end_index]
        lines.append(' '.join(word_list[start_index: end_index + 1]))
        #if we reached the first element its over
        if start_index == 1:
            break

    #printing from last line to first line
    for i in range(len(lines)-1, -1, -1):
        print(lines[i])
```

In [1]:
```
#this function print the given sentence to a well shaped paragraph
def word_wrapper(sentence, M):
    #iputs
    #sentence: a string of characters
    #M: maximum width of paragraph
    #output
    #print the sentence in a good shape

    #word extraction
    #list of individual words
        #first element is blank to start from index one
    word_list = [''] + sentence.split(' ')
    #list of words length
        #first element is zero to start from index one
    word_lentgh_list = np.array([len(word) for word in word_list])
    #number of all words
    n = len(word_lentgh_list) - 1

    #making cost matrix
        #first column and row are extra to start from index [1, 1]
    line_cost_matrix = np.ones([1 + n, 1 + n])
    for i in range(1, n + 1):
        for j in range(i, n + 1):
            #getting cost of putting i'th word to j'th word in one line
            line_cost_matrix[i, j] = line_cost_cal(word_lentgh_list, i, j, M)

    #making total cost array with dynamic prgramming
    #first element is zero
    C = [0]
    #keeping break points for print phase
    break_points = [0]
    #getting C[i] from C[i-1]
    for i in range(1, n + 1):
        #possible ways of breaking the words for last line with use of up-to i'th word
        candid_list = [(C[mid_index - 1] + line_cost_matrix[mid_index, i]) for mid_index in range(1, i + 1)]
        #selecting minimum for best solution up-to i'th word
        C.append(min(candid_list))
        #keeping the solution index as break index
        break_points.append(candid_list.index(min(candid_list)) + 1)

    #showing result
    #printing paragraph width with stars
    print('\n')
    print('*' * M)
    #calling "seq_printer" to show the result
    seq_printer(word_list, break_points)
    #printing paragraph width with stars
    print('*' * M)
```

In this part we set a sentence (intro of word wrapping) and make different values of M. Then we show result of function applied onthe sentence with different value of M.

In [2]:
```
#performing on a sequence
#parameters
#sentence: a long sentence
sentence = 'A sequence of words is given there is a limit on the number of characters for each line By putting line
 breaks in such a way that lines are printed clearly The lines must be balanced when some lines have lots of extra s
paces and some lines are containing a small number of extra spaces it will balance them to separate lines It tries t
o use the same number of extra spaces to make them balanced'
#M: paragraph width
M = [25, 35, 55, 75]

#testing
#different condition
for m in M:
    word_wrapper(sentence, m)
```

```
*************************
A sequence of words is
given there is a limit on
the number of characters
for each line By putting
line breaks in such a way
that lines are printed
clearly The lines must
be balanced when some
lines have lots of extra
spaces and some lines
are containing a small
number of extra spaces
it will balance them to
separate lines It tries
to use the same number of
extra spaces to make them
balanced
*************************


***********************************
A sequence of words is given
there is a limit on the number of
characters for each line By putting
line breaks in such a way that
lines are printed clearly The lines
must be balanced when some lines
have lots of extra spaces and some
lines are containing a small number
of extra spaces it will balance
them to separate lines It tries to
use the same number of extra spaces
to make them balanced
***********************************


*****************************************************
A sequence of words is given there is a limit on
the number of characters for each line By putting line
breaks in such a way that lines are printed clearly
The lines must be balanced when some lines have lots
of extra spaces and some lines are containing a small
number of extra spaces it will balance them to separate
lines It tries to use the same number of extra spaces
to make them balanced
*****************************************************


*******************************************************************************
A sequence of words is given there is a limit on the number of characters
for each line By putting line breaks in such a way that lines are printed
clearly The lines must be balanced when some lines have lots of extra
spaces and some lines are containing a small number of extra spaces it will
balance them to separate lines It tries to use the same number of extra
spaces to make them balanced
*******************************************************************************
```

Here we show time complexity of our wrapper function. As we said, brute force version is in order of exponentional. But with dynammic programming we do better. Making of $LineCostMatrix$ need $(1) + (1 + 2) + \ldots + (1 + \ldots + n)$ calculation which is $O(n^3)$. After that we build the C array, making this one needs $1 + 2 + \ldots + n$ calculation, so its in $O(n^2)$. With dynammic programming method we do wrapping task with time complexity of $O(n^3)$ and we need memory for $LineCostMatrix$ with order of $O(n^2)$.