CS 124: Problem Set 4

Chao Cheng
Collaborators: Steve Li, Elizabeth Ling

February 4, 2021

Problem 1. Let us the denote the maximum consecutive sum of any subarray of an array $A[1, 2, \ldots n]$ with n elements like this:

$$\text{MCS}(n) = \max_{1 \le i \le j \le n} \left(\sum_{k=i}^{j} A[k] \right)$$

In addition, we denote the maximum consecutive sum of any subarray ending at index n of $A[1 \ldots n]$:

$$\text{MCS}^*(n) = \max_{1 \le i \le n} \left(\sum_{k=i}^{n} A[k] \right)$$

If we can determine $\text{MCS}^*(k)$ for each $k = 1, 2, \ldots n$, then we will have determined $\text{MCS}(n)$ as well, since $\text{MCS}(n)$ is simply the largest of $\text{MCS}^*(1), \text{MCS}^*(2), \ldots \text{MCS}^*(n)$.

To determine $\text{MCS}^*(n)$, let us consider $\text{MCS}^*(n-1)$. We know that $\text{MCS}^*(n)$ has two possibilities: it is either $\text{MCS}^*(n-1)$ plus the n-th element, or it is just the n-th element. This is because $\text{MCS}^*(n)$ ends at n, and therefore must include n; so it will either consist of some sequence of elements before n (the best sequence being $\text{MCS}^*(n-1)$) plus $A[n]$, or it will just be $A[n]$, if $\text{MCS}^*(n-1) + A[n]$ is less than $A[n]$ alone. In other words, we have the following recursion:

$$\text{MCS}^*(n) = \max \left\{ \text{MCS}^*(n-1) + A[n], A[n] \right\}$$

Thus, in order to determine $\text{MCS}^*(n)$, we must determine $\text{MCS}^*(n-1)$, for which we must first determine $\text{MCS}^*(n-2)$, and so on until $\text{MCS}^*(1)$. Therefore, starting from index $i = 1$, we determine $\text{MCS}^*(i)$ (which for $i = 1$ is just $A[1]$, since there is only one possibility), and at each step, we use $\text{MCS}^*(i)$ determine $\text{MCS}^*(i+1)$ until we reach $\text{MCS}^*(n)$, keeping track of the largest value of $\text{MCS}^*(i)$ as well as its start and end indices. To do this, we initialize the start and stop indices to 1, and the largest value of MCS^* to negative infinity, and at each step, we compare $\text{MCS}^*(i)$ to the largest value and update it (and the indices) if needed. Thus, by the time we reach $\text{MCS}^*(n)$, we will have determined the largest $\text{MCS}^*(i)$ for $i = 1, 2, \ldots n$, which is equivalent to $\text{MCS}(n)$.

In the event that we determine $\text{MCS}(n)$ is negative, which would imply that all the elements of A are negative, we instead return 0 (i.e. the empty subarray).

The correctness of this algorithm is clear from induction, with the base case being $i = 1$, where we have $\text{MCS}^*(1) = \text{MCS}(i) = A[1]$. We can compute $\text{MCS}^*(i)$ from $\text{MCS}^*(i-1)$, which in turn means we can compute $\text{MCS}(i)$ for any $i \ge 1$.

The time complexity of this algorithm is $O(n)$, since we perform addition and comparison operations at each index i for a total of some constant times n times. The space complexity of this algorithm is $O(1)$, since we are only storing the largest value of MCS^* and its indices.

Problem 2. Let us define $M(n,k)$ to be the partitioning of an array A with n elements into $k+1$ subarrays using k indices with the minimal imbalance. We also define $\text{IMB}(A[a,b])$ of a subarray $A[a,b]$ as the absolute difference between its weight and the average weight of the subarrays:

$$\text{IMB}(A[a,b]) = \left| \sum_{i=a}^{b} \left(A[a] \right) - \frac{\sum_{i=1}^{n} A[i]}{k+1} \right|$$

In other words, $M(n,k)$ is the least maximum imbalance of any partitioning of A into $k+1$ subarrays.

To determine $M(n,k)$, suppose the k indices $j_1, j_2 \dots j_k$ partition the size n array into subarrays $A[1,j_1], A[j_1+1,j_2], \dots A[j_k+1,n]$, and consider the last index, j_k. Suppose that we place j_k at index i. Then the imbalance of this partitioning has two possibilities. Either it is $\text{IMB}(A[i+1,n])$, or it is the maximum imbalance of any of the subarrays to the left of i, namely $A[1,j_1], A[j_1+1,j_2], \dots, A[j_{k-1}+1,i]$; whichever is larger. Thus, to minimize the partitioning, we would want to minimize the imbalances of the subarrays to the left of i, which is equivalent to $M(i,k-1)$, the problem of finding the minimal partitioning of the array $A[1,i]$ into k subarrays using $k-1$ indices. In mathematical form, this recursive relationship looks like:

$$M(n,k) = \begin{cases} \min_{k \le i \le n} \left(\max \left\{ \text{IMB}(A[i+1,n], M(i,k-1) \right\} \right) & \text{if } k \ge 1 \\ 0 & \text{if } k = 0 \end{cases}$$

where we check each selection of i starting from index k (since the $k-1$ subarrays to the left of i will at least result in partition $k-1$ ending at index $k-1$) until n to find the i that results in the minimal imbalance across all subarrays. Our base case for the recursion is when $k=0$ (since the minimal imbalance of 1 subarray, which is the same as the base array, is 0).

The dynamic programming approach to solving this problem is to start from the bottom and build up, storing our results in a $n \times k$ array B where $B[i,j] = M(i,j)$ and filling each row from left to right and the columns from top to bottom. We also store the result of $\text{IMB}(A[j+1,n])$ in a variable to make computing the next entry in B more efficient. Finally, we also track the index positions to obtain the minimal partitioning in another $n \times k$ matrix D, storing the last index i in $D[a,b]$ from the minimal partitioning of $M(a,b)$. In the end, we reconstruct the indices of the minimal partitioning of $M(n,k)$ by tracing backwards in D, so that $j_k = D(n,k)$, $j_{k-1} = D(j_k, k-1)$, and so on until we obtain $j_1 = D(j_2, 1)$, adding each index to an array and returning the array of indices $j_1 \dots j_k$ at the end.

The correctness of this algorithm is clear from induction, using the base case of $k=0$ and constructing B until we arrive at $B[n,k] = M(n,k)$.

The time complexity of this algorithm is $O(kn^2)$, since it takes at most $O(n)$ calculations to compute each entry of the DP array, B (because we have recorded the imbalances from previous entries), and there are $O(kn)$ entries in the array. The space complexity of this algorithm is $O(kn)$, because the sizes of B and D are both kn.

If the imbalance of a partitioning was instead changed to the sum of the imbalances of each subarray, then to determine $M(n,k)$, we would instead consider the placement of i for the j_k partition that results in the minimum overall imbalance, or in other words, the smallest sum of $\text{IMB}(A[i+1,n])$ plus the sum of the imbalances of any of the subarrays to the left of i. In mathematical form:

$$M(n,k) = \min_{k \le i \le n} \left\{ \text{IMB}(A[i+1,n]) + M(i,k-1) \right\}$$

Since we have simply redefined the recursive relationship of $M(n,k)$, the rest of the algorithm remains the same so we have the same time and space complexity.

Problem 3. In this algorithm, we use a data structure to represent the tree T wherein each node stores pointers to its children in the tree. We define $\texttt{minblanket}(n)$ to be the size of the minimum blanket of the tree T with root node n, and denote the k children of n as $n_1, n_2, n_3, \ldots n_k$. Additionally, we initialize an array A of size n to store the nodes of the minimum blanket. Let us begin by considering the root node n. There are two possibilities: n is in $\texttt{minblanket}(n)$, or it is not in the minimum blanket.

If n is in $\texttt{minblanket}(n)$, then we have the following recursion (given that n has k children):

$$\texttt{minblanket}(n) = 1 + \sum_{i=1}^{k} \texttt{minblanket}(n_k)$$

In other words, the size of the minimum blanket of T is the sum of the sizes of the subtrees starting from the children of the root node, plus 1 (since the root node itself is in the minimum blanket).

If n is not in $\texttt{minblanket}(n)$, this would imply that the edges between n and its children are already in $\texttt{minblanket}(n)$, which in turn implies that all the children of n are in $\texttt{minblanket}(n)$. If this is the case, then we have the following recursion (given that each child n_i has k_i children of its own):

$$\texttt{minblanket}(n) = k + \sum_{i=1}^{k} \sum_{j=1}^{k_i} \texttt{minblanket}(n_{i,j})$$

where $n_{i,j}$ denotes the j-th child of the i-th child of n. In other words, the size of the minimum blanket is the sum of the sizes of the subtrees starting from the grandchildren of n, plus k (since the k children of n are all in the minimum blanket of T.

Thus, we have that the minimum blanket of the tree with root node n is the minimum of these two possibilities:

$$\texttt{minblanket}(n) = \begin{cases} \min\left\{1 + \sum_{i=1}^{k} \texttt{minblanket}(n_k), k + \sum_{i=1}^{k} \sum_{j=1}^{k_i} \texttt{minblanket}(n_{i,j})\right\} & \text{if } n \text{ has children} \\ 0 & \text{if } n \text{ has no children} \end{cases}$$

If n has children, then when we compare the two possibilities, if the minimum comes from n being in the minimum blanket, then we add n to the minimum blanket array A; otherwise we do not add it. The base case is when we reach a node has no children (i.e. a leaf node), in which case we return 0, since this node will not be in the minimum blanket.

To avoid solving the same problem multiple times, the dynamic programming approach is that we store the size of $\texttt{minblanket}(a)$ in node a and determine $\texttt{minblanket}(n)$ by first determining the minimum blanket sizes for the base cases (the leaf nodes, which will not be in the minimum blanket of T) and then recursively calculating the $\texttt{minblanket}$ for their parents and so on until we reach $\texttt{minblanket}(n)$, adding nodes to A as we go.

The correctness of this algorithm is clear from induction, using the base case of leafless nodes and building up to the root node.

Since the algorithm visits each node at least twice and otherwise performs only addition operations, the time complexity of this algorithm is $O(n) = O(|V|)$ for a tree with n nodes. Since it stores the $\texttt{minblanket}$ values in the nodes of the tree, and the nodes of minimum blanket in an array of size at most n, the space complexity is $O(n) = O(|V|)$.

Problem 4. We define a two-dimensional array, `cost`, where the entry `cost[i,j]` is defined as:

$$\texttt{cost[i,j]} = \left(M - j + i - \sum_{k=i}^{j} \ell_k\right)^3$$

In other words, `cost[i,j]` is the cost of putting words ℓ_i through ℓ_j in a single line. If ℓ_i through ℓ_j will not fit in a single line (i.e. if `cost[i,j]` is negative), then we instead set `cost[i,j]` $= \infty$ to avoid using it in our final solution. Also, in the case that $j = n$, we set `cost[i,n]` $= 0$ for all i because we don't consider the cost of the last line.

We also define $\texttt{C}(n)$ to be the optimal (minimal) cost of arranging words ℓ_1 to ℓ_n. In order to minimize the cost of arranging words ℓ_1 to ℓ_n, suppose the last line begins with ℓ_k and continues until ℓ_n. Then the total cost of this arrangement is $\texttt{C}(k-1) + \texttt{cost[k,n]}$. If $k = n$, then we are simply taking $\texttt{C}(n-1)$ and adding the cost of adding ℓ_n to a new line; otherwise, when $k < n$, we are taking $\texttt{C}(k-1)$ and adding the cost of adding some words ℓ_k to ℓ_n to the last line. Thus, we have the following recursion:

$$\texttt{C}(n) = \min_{1 \le k \le n} \left\{\texttt{C}(k-1) + \texttt{cost[k,n]}\right\}$$

Thus, to compute $\texttt{C}(n)$ we must first compute $\texttt{C}(n-1), \texttt{C}(n-2), \ldots \texttt{C}(1)$. Thus, the dynamic programming approach is to start from $\texttt{C}(1)$ (which is just `cost[1,1]`, since $\texttt{C}(0) = 0$) and build our way up to $\texttt{C}(n)$, storing the results in an array.

In order to keep track of the arrangement of words, we also maintain a separate array `B` where `B[n]` records the index k of the first word of the last line of $\texttt{C}(n)$, so that when we reconstruct our paragraph, the last line starts with word $\ell_{\texttt{B[n]}}$ and ends with ℓ_n, the second to last line starts with word $\ell_{\texttt{B[n-1]}}$ and ends with $\ell_{\texttt{B[n]-1}}$, and so on until the first line, which starts with $\ell_{\texttt{B[1]}} = \ell_1$ and ends with $\ell_{\texttt{B[2]-1}}$.

The correctness of this algorithm is clear from induction, with the base cases being $\texttt{C}(0) = 0$ and $\texttt{C}(1) = \texttt{cost[1,1]}$.

The time complexity of this algorithm is $O(n^2)$, since for each of the n words, we evaluate $O(n)$ scenarios to determine the minimum cost of adding that word to the optimal paragraph. The space complexity is also $O(n^2)$, since we store the costs of putting words i through j in a two-dimensional array. We can improve this to $O(bn)$ for some constant $b < n$ by, for each $\texttt{C}(n)$, calculating $\min\left\{\texttt{C}(k-1) + \texttt{cost[k,n]}\right\}$ only for k between b and n (where $\ell_b \ldots \ell_n$ is the maximum amount of words that can fit on the final line) instead of calculating it for any k between 1 and n, since any word before ℓ_b will not be able to fit on the last line and thus should not be considered.

For $M = 40$, the penalty is 2183, and for $M = 72$, the penalty is 2104. The output of the algorithm is copied below, and code is in the appendix at the end.

```
----------------------------------------
Source file:  buffy.txt
M: 40
Buffy the Vampire Slayer fans are
sure to get their fix with the DVD
release of the show's first season.
The three-disc collection includes all
12 episodes as well as many extras.
There is a collection of interviews
by the show's creator Joss Whedon in
```

```
which he explains his inspiration for
the show as well as comments on the
various cast members.  Much of the same
material is covered in more depth with
Whedon's commentary track for the show's
first two episodes that make up the
Buffy the Vampire Slayer pilot.  The
most interesting points of Whedon's
commentary come from his explanation
of the learning curve he encountered
shifting from blockbuster films like Toy
Story to a much lower-budget television
series.  The first disc also includes
a short interview with David Boreanaz
who plays the role of Angel.  Other
features include the script for the
pilot episodes, a trailer, a large photo
gallery of publicity shots and in-depth
biographies of Whedon and several of the
show's stars, including Sarah Michelle
Gellar, Alyson Hannigan and Nicholas
Brendon.
Penalty:  2183
---------------------------------------
Source file:  buffy.txt
M: 72
Buffy the Vampire Slayer fans are sure to get their fix with the
DVD release of the show's first season.  The three-disc collection
includes all 12 episodes as well as many extras.  There is a collection
of interviews by the show's creator Joss Whedon in which he explains
his inspiration for the show as well as comments on the various cast
members.  Much of the same material is covered in more depth with
Whedon's commentary track for the show's first two episodes that make
up the Buffy the Vampire Slayer pilot.  The most interesting points of
Whedon's commentary come from his explanation of the learning curve
he encountered shifting from blockbuster films like Toy Story to a
much lower-budget television series.  The first disc also includes a
short interview with David Boreanaz who plays the role of Angel.  Other
features include the script for the pilot episodes, a trailer, a large
photo gallery of publicity shots and in-depth biographies of Whedon and
several of the show's stars, including Sarah Michelle Gellar, Alyson
Hannigan and Nicholas Brendon.
Penalty:  2104
```

Problem 4 Code

```
# Written in Python 3.7
def optimalCost(words, M):
    n = len(words) - 1  # Don't include the placeholder in number of words
    lengths = [len(w) for w in words]
    inf = float("inf")

    # Initialize helper arrays
    # Stores the extra spaces in a line containing words i through j
    spaces = [[0 for i in range(n+1)] for i in range(n+1)]
    # Stores the cost of a line containing words i through j
    cost = [[0 for i in range(n+1)] for i in range(n+1)]
    # Stores the indices of the first word of each line in the optimal paragraph
    B = [0 for i in range(n+1)]
    # Stores the optimal cost of the first i words
    C = [0 for i in range(n+1)]

    # Populate spaces array
    for i in range(1, n+1):
        spaces[i][i] = M - lengths[i]
        for j in range(i+1, n+1):
            spaces[i][j] = spaces[i][j-1] - lengths[j] - 1

    # Populate cost array
    for i in range(1, n+1):
        for j in range(i, n+1):
            if spaces[i][j] < 0:  # If the words don't fit in one line, the cost is
                cost[i][j] = inf
            elif j == n:  # We don't consider the cost of the last line
                cost[i][j] = 0
            else:
                cost[i][j] = spaces[i][j] ** 3

    # Base cases for optimal cost
    C[0] = 0
    C[1] = cost[1][1]
    for i in range(2, n+1):  # Calculate entries of C from left to right
        C[i] = inf
        for k in range(1, i+1):
            if C[k-1] != inf and cost[k][i] != inf:
                if C[k-1] + cost[k][i] < C[i]:
                    C[i] = C[k-1] + cost[k][i]
                    B[i] = k

    return printParagraph(n, B, words, cost)
```

```
# Recursively calculate the total penalty and print out each line of the optimized
def printParagraph(n, B, words, cost):
    penalty = 0
    if B[n] != 1:
        penalty = printParagraph(B[n] - 1, B, words, cost)
    line = " ".join(words[B[n]:n+1])
    print(line)
    penalty += cost[B[n]][n]
    return penalty


if __name__ == "__main__":
    source_file = "buffy.txt"
    words = [""]  # Placeholder at index 0 so that words start from index 1

    with open(source_file, "r") as f:  # Get list of words from source file
        for line in f:
            for word in line.split():
                words.append(word)

    for m in [40, 72]:
        print(f"----------------------------------------")
        print(f"Source file: {source_file}")
        print(f"M: {m}")
        penalty = optimalCost(words, m)
        print(f"Penalty: {penalty}")
```