CSCI 5454: Algorithms: Homework 6

Ashutosh Gandhi

April 8, 2023

Problem 1

1.1 Part A

The Recurrence For the Algorithm is as follows

if the node is not a leaf:
maxWeightIncluding(node) = node.weight + maxWeightNotIncluding(node.left) + maxWeightNotIncluding(node.right)

maxWeightNotIncluding(node) = max(maxWeightIncluding(node.left), maxWeightNotIncluding(node.left))
+ max(maxWeightIncluding(node.right), maxWeightNotIncluding(node.right))

if the node is a leaf:
maxWeightIncluding(node) = node.weight
maxWeightNotIncluding(node) = 0
Where node.left and node.right are the left and right children of any node passed to the function.

pseudo-code for the memo table is

```
class Node:
    def __init__(self):
        self.weight = 0
        self.maxWeightIncluded = -math.inf
        self.maxWeightExcluded = -math.inf
        self.left = self.right = None


def maxWeightNotIncluding(node):
    if node == None:
        return 0
```

```
    if node.maxWeightExcluded != -math.inf: # make use of the memo table
        return node.maxWeightExcluded

    if node.left == None and node.right == None: # leaf node base case
        node.maxWeightExcluded = 0
        return node.maxWeightExcluded

    node.maxWeightExcluded = max(maxWeightIncluding(node.left),
    ↪  maxWeightNotIncluding(node.left)) +
    ↪  max(maxWeightIncluding(node.right),
    ↪  maxWeightNotIncluding(node.right)) # recurse
    return node.maxWeightExcluded

def maxWeightIncluding(node):
    if node == None:
        return 0

    if node.maxWeightIncluded != -math.inf: # make use of the memo table
        return node.maxWeightIncluded

    if node.left == None and node.right == None: # leaf node base case
        node.maxWeightIncluded = node.weight
        return node.maxWeightIncluded

    node.maxWeightIncluded = node.weight + maxWeightNotIncluding(node.left)
    ↪  + maxWeightNotIncluding(node.right) # recurse
    return node.maxWeightIncluded

def main():
    root = create_tree()
    maxWeight = max(maxWeightIncluding(root), maxWeightNotIncluding(root))
    print(maxWeight)
```

To make sure that the computation of max weight is not repeated at any node, we can augment the maxWeightIncluded and maxWeightExcluded values for each node in the tree and add an extra base condition to return the value if already present. This way we don't recurse over the children of a node to re-compute its max weight. The node representation and the 2 recursive function codes are shown above. We start with checking the weights of including or not including the root node and recurse from there. the maxWeight variable above gives the maximum weight of an independent set of the tree.

Now, to print the sequence of nodes selected, we can make a pre-order tree traversal and include a node if its Include-Weight is more than its Exclude-Weight. Then, if the node gets included then skip the children and go to its grandchildren otherwise if not included then to

its children to make the check the Include, Exclude weights.

pseudo-code to recover the independent set of nodes is

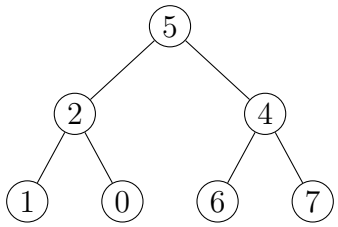
```
def get_node_set(root: Node, ans: list):
    if root: # perform pre-order tree traversal
        if root.maxWeightIncluded > root.maxWeightExcluded: # include node
        ↪  and go to grandchildren
            ans.append(root.weight)
            if root.left != None and root.right != None: # only traverse if
            ↪  not a leaf node
                get_node_set(root.left.left, ans)
                get_node_set(root.left.right, ans)
                get_node_set(root.right.left, ans)
                get_node_set(root.right.right, ans)
        else: # go to children
            get_node_set(root.left, ans)
            get_node_set(root.right, ans)
    return ans

node-set = get_node_set(root, [])
```

The overall running time of the complete algorithm is $O(n)$, where n is the number of nodes in the tree. Initially, we make one pass over the tree to compute the weights at each node and another pass to get the sequence of nodes in the independent set.

1.2 Part B

Consider the following binary tree



Using the Greedy algorithm we get the node-set as $\{2, 6, 7\}$ with the maxWeight as 15.
Using the DP algorithm we get the node-set as $\{5, 1, 6, 7\}$ with the maxWeight as 19.

Thus the Greedy algorithm gives a sub-optimal solution for the above tree compared to the DP algorithm.

Problem 2

2.1 Part A

The Recurrence for the Algorithm is as follows

$$\text{MinPalindromeIns(i,j)} = \begin{cases} \text{MinPalindromeIns(i+1,j-1)} & \text{, if s[i]==s[j]} \\ \min(\text{MinPalindromeIns(i+1,j)}, \text{MinPalindromeIns(i,j-1)}) & \text{, if s[i]!=s[j]} \end{cases}$$

With the base case of returning 0 for when i==j, strings of length 0 and 1.

2.2 Part B

pseudo-code for the memo table for the above recurrence is:

```
def MinPalindromeIns(s: str, i: int, j: int, mem_table: list[list]):

    if i > j: # in case the pointers cross over resulting in a substring
    ↪  size less than 0
        return math.inf
    if i == j: # string of len 1 is a palindrome
        return 0
    if i == j-1:
        if s[i] != s[j]:
            mem_table[i][j] = 1
        return mem_table[i][j]

    if s[i] == s[j]: # first and last element is the same
        mem_table[i][j] = MinPalindromeIns(s, i+1, j-1, mem_table)
    else:
        mem_table[i][j] = 1 + min(MinPalindromeIns(s, i+1, j, mem_table),
        ↪  MinPalindromeIns(s, i, j-1, mem_table))

    return mem_table[i][j]
```

To construct the memo table, we store the outputs given by the recurrence defined above into a 2D matrix of size N*N. if the same substring is to be computed then we can directly look up the table instead of computing the value recursively. The base condition is a sub-string of size 0 and 1 which are already palindrome so we return 0.
The minimum number of insertions required can be gotten by reading the value at mem_table[0][n-1]. This value can be accessed in $O(1)$ time. The algorithm would fill the top right half of the matrix and hence the overall time to fill the memo table is $O(n^2)$.

2.3 Part C

code for retrieving the string from the memo table is:

```
def get_palindrome(s: str, mem_table: list[list]):
    n = len(s)
    i, j = 0, n-1
    result = list(s)
    append_list = []
    prepend_list = []
    while i < j:
        if s[i] == s[j]:
            i += 1
            j -= 1
        elif mem_table[i][j-1] <= mem_table[i+1][j]: # s[j] to be inserted
        ↪  to the beginning of the sub-string
            prepend_list.append((i, s[j]))
            j -= 1
        else: # s[i] to be inserted to end of the sub-string
            append_list.append((j+1, s[i]))
            i += 1
    # form the palindrome string using the append and prepend lists
    for ind, char in append_list:
        result.insert(ind, char)
    for ind, char in reversed(prepend_list):
        result.insert(ind, char)

    return "".join(result)
```

The Steps to retrieve palindrome string are

- we backtrack from position [0][n-1] in the mem_table to recover the insertion character and position of insertion.
- we compare the [i][j-1] value with [i+1][j], if it is less then we append the character s[j] to the end of the sub-string otherwise s[i] needs to be prepended to start of the sub-string. In case the character i and j is the same then go to i+1,j-1 (similar step as done in recurrence for memo-table formation)
- The above step is repeated till i remain less than j. The character to be inserted and the position to insert is stored in 2 separate lists depending on whether it has to be appended or prepended. The final string is formed by inserting the characters in these 2 lists into the original string.

The running time for this retrieval is $O(n^2)$, thus the overall time for the complete Algorithm is $O(n^2)$.