Yunjia Zeng
yz682

Homework 5

ANLY 550
April 27, 2019

Problem 1

Proof. Suppose there are T_1 and T_2, which are both MST of a graph G, and has equal weights. Let $e_1 \in T_1$ be an edge and $e_1 \notin T_2$. Then $T_2 \cup \{e_1\}$ must contain a cycle, and there exist an edge e_2, in this cycle, is not in T_1. Then if $T_2 \cup \{e_1\} - \{e_2\}$ which has a smaller weight than T_2, but T_2 is a minimum spanning tree. Therefore, we proved by contradiction. □

Problem 2

This problem is actually finding the maximum subarray in array A. We can solve it by dynamic programming through the following algorithm.

Algorithm 1: Find Maximum Subarray

```
global_maximum = local_maximum = A[0]
for x in A[1:]:
    local_maximum = max(local_maximum + x, x)
    global_maximum = max(global_maximum, local_maximum)
return global_maximum
```

In each step, we check if the current element plus the last largest sum (local_maximum) is greater than the current maximum (global_maximum). If yes we update the local_maximum, otherwise the current element is the largest subarray (therefore the subarray size is 1). Then we update the global_maximum if there is a new global maximum. Whenever a sub-sequence is encountered which has a negative sum, the next sub-sequence to examine can begin after the end of the sub-sequence which produced the negative sum. In other words, there is no starting point in that sub-sequence which will generate a positive sum. Therefore, they can all be ignored. We prove the correctness of this algorithm by induction.

The runtime of this algorithm is $O(n)$, since it only loops through the array once and contains only constant time operations.

Problem 3

We first recursively define the value of the optimal solution. Let $c(j)$ be the optimal cost of printing words 1 through j. Given the index of the first word printed on the last line of an optimal solution, we have

$$c(j) = c(i-1) + linecost(i, j).$$

But since we do not know what i is optimal, we need to consider every possible i so that the recursive definition of the optimal cost is

$$c(j) = min_{1 \leq i \leq j}\{c(i-1) + linecost(i,j)\}.$$

To simplify the problem, we defined $c(0) = 0$, and calculate the values of the array c from index 1 to n, bottom up. We adopt p, to keep track of the actual optimal arrangement of the words, where $p(k)$ is the i which led to the optimal $c(k)$. Then after arrays for c and p computed, the optimal cost is $c(n)$ and the optimal solution can be found by printing words $p(n)$ through n on the last line, words $p(p(n) - 1)$ through $p(n) - 1$ on the next to last line, and so on. We define an array L such that

$$L[0] = 0$$
$$L[i] = L[i-1] + l_i \equiv \sum_{k=1}^{i} l_k,$$

where $L[i]$ is a cumulative sum of lengths of words 1 through i. The output of this problem is actually a sequence of integers $< i_1, i_2, \ldots, i_k$, where $i \leq i_1 \leq i_2 \leq \ldots \leq i_h = n$. Therefore i_j is the index of the last word appearing on line j, for $1 \leq j \leq h$. The possible output sequence is valid if and only if

$$i_j - (i_{j-1} + 1) + \sum_{k=i_{j-1}+1}^{i_k} l_k \leq M,$$

where $i_{-1} = 0$ for $1 \leq j \leq h$. This ensure the words on any given line fit into the space available for them. To better explain how to solve this problem, we define E as the extra space remaining at the end of a line containing words i through j:

$$E[i,j] \coloneqq M - j + i - (L[j] - L[i-1]).$$

Note that E could be negative. Now we define C - the cost of including a line containing words i through j - as a sum that we are to minimize:

$$linecost(i,j) = \begin{cases} \infty & \text{if E[i,j]} < 0, \\ 0 & \text{if j = n and E[i,j] } \leq 0, \\ E[i,j]^3 & \text{otherwise} \end{cases}$$

Algorithm 2: Print-Neatly(l,n,M)

```
let E[i..n, 1..n], C[1..n, 1..n], and c[0..n] be new arrays
for i = 1 to n:
    E[i,i] = M - li
    for j = i + 1 to n
        E[i,j] = E[i, j-1] - lj - 1
    for i = 1 to n
```

```
        for j = i to n
            if E[i,j] < 0
                C[i,j] = infinity
                else if j == n and E[i,j] >= 0
                    C[i,j] = 0
                else C[i,j] = E[i,j]**3
c[0] = 0
for j = 1 to n
    c[j] = infinity
    for i = 1 to n
        if c[i-1] + C[i,j] < c[j]
            c[j] = c[i - 1] + C[i,j]
            p[j] = i
return c, p
```

This algorithm takes $O(n^2)$ time, since each value of c takes up to n calculations as each value of i is considered. But by noticing that at most $\lfloor (M+1)/2 \rfloor$ words can fit on a single line, we could reduce the running time to $O(nM)$, by considering only the is for which

$$j - \lfloor (M+1)/2 \rfloor + 1 \leq i \leq j,$$

when calculating each c(j). Since a line with words $i, \ldots, j$ contains $j - i + 1$ words, if $j - i + 1 > \lfloor (M+1)/2 \rfloor$ then we know that $linecost(i, j) = \infty$. We only need to compute and keep track of $E[i, j]$ and $linecost(i, j)$ for $j - i + 1 \leq \lfloor (M+1)/2 \rfloor$. And for the inner for loop that computes the $c[j]$ and $p[j]$ can be run from $max(1, j - \lfloor (M+1)/2 \rfloor + 1)$ to j.

Problem 4

For this problem, we want to find the maximum-sized independent set S in a graph $G = (V, E)$ that is a tree with root r, for short we call this tree T. We can apply depth first search to find S in T. We will search each sub-tree and calculate two value:

1. A(i), which is the size of the maximum-sized independent set among all sub-trees that rooted in i, such that i is included in this independent set.

2. B(i), which is the size of the maximum-sized independent set among all sub-trees that rooted in i, such that i is not included in this independent set.

Therefore, we will have a recursion under two cases:

1. When i is not an element of the so far maximum-sized independent set, we have $B(i)$ as the union of the i's children. Therefore $B(i) = max(A(j), B(j))(i))$, such that j represents all children nodes at i's level.

2. When i is an element of the so far maximum-sized independent set, we have $A(i)$ as the union of i and the maximum-sized independent set of i's grandchildren. Therefore, $A(i) = 1 + sum(B(j))$ for every j in i.

From above, we derive the recursive equation as follows. Let $S(i)$ be the maximum independent set in the sub-tree rooted at vertex i:

$$size(S(i)) \coloneqq max\{\sum_{\text{j child of i}} size(S(i)), 1 + \sum_{\text{j grandchild of i}} size(S(i))\}.$$

Algorithm 3: MaxIndSet(r)

```
initialize tables a and b with all zeros
sumA = 0
sumB = 0
for each non visited i neighbor node do:
    MaxIndSet(i)
    sumA = sumA + b['i']
    sumB = sumB + max(a['i'], b['i'])
endfor
a['r'] = 1 + sumA
b['r'] = sumB
output max(a['r'], b['r'])
```

The element $a[i]$ is the size of S in the sub-tree rooted in i such that $i \in S$, and $b[i]$ is the size of S in the sub-tree rooted in i such that $i \notin S$. Since for each vertex, the algorithm only looks at its sub-trees, i.e. the children and grandchildren; therefore, each vertex i will be visited when $MaxIndSet$ is processing vertex i, when $MaxIndSet$ is processing i's parent, and when $MaxIndSet$ is processing i's grandchildren. Therefore, i is looked only three times, which is constant operation, hence the total number of steps is in $O(n)$ since there are n entries to be filled in.