CS 180 - Homework 4

Darren Tsang, Discussion 1I

Produced on Thursday, Jun. 04 2020 @ 04:36:06 PM

Question 1

Part A

	day1	day2	day3	day4	day5
x	100	100	100	100	100
s	99	5	4	3	2

The optimal solution would be to reboot on day2 and day4, which yields a total of $99+0+99+0+99 = 297$ terabytes.

Part B

processedTB($X = [x_1, x_2, ..., x_n]$, $S = [s_1, s_2, ..., s_n]$){
    solverTable = n by n table
    solverTable[n, j] = min(x_n, s_j) for $j = 1, ..., n$
    for $i = (n-1), ..., 1$ {
        for $j = 1, ...i$ {
            reboot = solverTable[i + 1, 1]
            continue = min(x_i, s_j) + solverTable[i + 1, j + 1]
            solverTable[i, j] = max(reboot, continue)
        }
    }
    return solverTable[1, 1]
}

The algorithm above takes a top down approach to determines the maximum total TB that can be processed given $x_1, x_2, ..., x_n$ and $s_1, s_2, ..., s_n$. We create *solverTable*, where $solverTable[i, j]$ is the maximum amount of processed TB for days i to n and j is days since last reboot. On each day, we have two decisions, to either reboot or continue, which is why I calculated *reboot* and *continue*. Since we are trying to maximize TB processed, I set $solverTable[i, j]$ to be the max of *reboot* and *continue*. Also, I take advantage of the fact that rebooting on day n is pointless by setting $solverTable[n, j] = \min(x_n, s_j)$ for $j = 1, ..., n$.

Since there are two nested for loops that rely on n, the algorithm is $O(n^2)$.

Question 2

```
createPalindrome(String s, int n){
    solverTable = n x n table
    solverTable = 0 for all entries

    for i = 1, ..., n {
        start = 0
        for end = i, ..., n {
            if s[start] == s[end] {
                solverTable[start, end] = solverTable[start + 1, end - 1]
            } else {
                solverTable[start, end] = 1 + min(solverTable[start, end - 1], solverTable[start + 1, end])
            }
            start = start + 1
        }
    }
    return solverTable[0, n-1]
}
```

To start off, we create an n x n table initialized to be 0 to store our results. solverTable[i,j] represents the insertions to form a palindrome from the i^{th} to j^{th} charcter. When $s[start] == s[end]$, the number of insertions will not be affected by s[start] and s[end], which is why we set $solverTable[start, end] = solverTable[start + 1, end - 1]$. When $s[start] \neq s[end]$, we know an insertion is needed, which explains the + 1. Additionally, we are trying to find the minimum additions needed, which is why we are taking the minimum between $solverTable[start, end - 1]$ and $solverTable[start + 1, end]$. Finally, $solverTable[0, n - 1]$ is returned as our final solution.

Since there are two nested for loops that rely on n, the algorithm is $O(n^2)$.

Question 3

```
findMCIS(Node n, Set X){ \\ MCIS = Maximum Cardinality Independent Set
    if(n is null){
        return 0
    } else if(n is a leaf){
        X = X ∪ {n}
        return 1
    }

    left = left child of n
    leftleft = left child of left
    leftright = right child of left

    right = right child of n
    rightleft = left child of right
    rightright = right child of right

    including = findMCIS(leftleft, X) + findMCIS(leftright, X) + findMCIS(rightleft, X) + findMCIS(rightright, X) + 1
    excluding = findMCIS(left, X) + findMCIS(right, X)

    if(including > excluding){ X = X ∪ {n}}
    return(max(including, excluding))
}

X = empty set
count = findMCIS(root, X)
return X, count
```

When findMCIS(n, X) is called, there are two scenarios we must consider: n is part of an MCIS of T or n is not part of an MCIS of T.

To account for the first scenario, *including* is calculated by calling findMCIS() on the children of the children of n. We know that we want an independent set, so we can disregard the children of n because that would lead to a contradiction. Note that *including* has + 1 at the end because we are including node n.

To account for the second scenario, *excluding* is calculated by calling findMCIS() on the children of n. Since n is not being included (aka it is excluded), there is no contradiction like the one in the first scenario.

findMCIS() will really only have two differnt options for what is returned. findMCIS() will return 0 if n is null because there is no node, else max(including, excluding) will be returned because we are trying to find a MaximumCIS. Note that when n is a leaf, $including = 0 + 0 + 0 + 0 + 1 = 1$ and $excluding = 0 + 0 = 0$, thus $including > excluding$; I take advantage of the fact that $max(including, excluding) = including = 1$ to directly return 1 without having to actually calculate *including* and *excluding*.

Note that X will be updated accordingly depending on whether *including* or *exlcuding* is larger.

The three lines at the very end of the peusdocode can just be thought of as a driver. It creates an empty set, runs findMCIS(), and returns X and *count*.

For each node (that is not the root) n, n is visited a constant number of times: when n is a child and when n is a child of a child. The algorithm visits all nodes a constant number of time, thus the algorithm is $O(|V|)$.

Question 4

We begin by changing any instance of vertex cover problem into an instance of the hitting set problem. Let $G = (V, E), k$ be an instance of vertex cover. Let $A = V$. Then, order the edges $\in$ E in any way to obtain the ordering $e_1 = (u_1, v_1), e_2 = (u_2, v_2), ..., e_{|E|} = (u_{|E|}, v_{|E|})$. Let $B_i = \{u_i, v_i\}$ for all $i \leq |E|$. Now, the instance of vertex cover has been changed into an instance of the hitting set problem.

Claim: There is a hitting set H such that $|H| \leq k$ if and only if the graph G has a vertex cover C such that $|C| \leq k$.

($\rightarrow$) Assume that there is a hitting set H such that $|H| \leq k$. Since H is a hitting set, $H \cup B_i \neq \emptyset$ for all $i \leq |E|$. This means that $H \cup \{u_i, v_i\} \neq \emptyset$ for all $i \leq |E|$. Thus, H is a vetex cover such that $|H| \leq k$, as desired.

($\leftarrow$) Assume that the graph $G = (V, E)$ has a vertex cover C such that $|C| \leq k$. Then, it is true that for all $e_i = (u_i, v_i)$ such that $i \leq |E|$, we know that $u_i \in C$ or $v_i \in C$, which means that $B_i \cup C \neq \emptyset$. Thus, C is a hitting set such that $|C| \leq k$, as desired.

Question 5

I will be taking the approach suggested on Page 490 of our course textbook, *Algorithm Design*.

We begin by changing any instance of a 3-colorable problem into an instance of a 4-colorable problem. Let $G = (V, E)$ be an instance of a 3-colorable problem. Then, create a new node $v' \notin V$ and a set $E' = \{e = (v', w), \forall w \in V\}$. Then, let $G' = (V \cup v', E \cup E')$ be an instance of a 4-colorable problem.

Claim: G is 3-colorable if and only if G' is 4-colorable.

($\rightarrow$) Assume that $G = (V, E)$ is 3-colorable. Then, we set the new node v' to a new, unique color c, meaning that v' is the only node with color c. Using the fact that G is 3-colorable and v' is color c, we can see that G' is 4-colorable, as desired.

($\leftarrow$) Assume that $G' = (V \cup v', E \cup E')$ is 4-colorable. Since G' is 4-colorable and v' is connected to all other nodes, we know that v' has its own unique color c. Then, we remove v' and edges that have v' as an end to obtain G. After removing the color c, we can see that G must be 3-colorable, as desired.