CS180-Spring20 Homework4

Gurbir Singh Arora

TOTAL POINTS

95 / 100

QUESTION 1

1 computing system 25 / 25

✓ **- 0 pts Correct**

- 2 pts minor mistake

- 3 pts a: not stating the optimal solution

- 20 pts b: Incorrect algorithm

- 10 pts b: Algorithm not optimal

- 10 pts b: major mistake

- 5 pts a) incorrect

- 5 pts b: not complete

QUESTION 2

2 palindrome 25 / 25

✓ **- 0 pts Correct algorithm**

- 25 pts Incorrect algorithm

- 8 pts Missing Psuedocode

- 5 pts No explanation of runtime or incorrect runtime

- 5 pts No explanation of correctness of algorithm

QUESTION 3

3 IS 25 / 25

✓ **- 0 pts Correct**

- 25 pts incorrect

- 10 pts major mistake

- 15 pts Solution incomplete

- 5 pts minor mistake

QUESTION 4

4 Hitting set 10 / 10

✓ **- 0 pts Correct**

- 10 pts Incorrect

- 5 pts proof not clear enough

QUESTION 5

5 K-coloarable 10 / 15

- 0 pts Correct

- 10 pts 1. Did not prove that G is 3-colorable if and only if G' is 4-colorable.

✓ **- 5 pts 2. Did not prove that G' can be computed from G in polynomial time.**

Gurbir Arora
105178554

CS 180 HW #4

1) Given the amounts of available data $x_1, x_2, \ldots, x_n$ for the next n days, and given the profile of your system as expressed by $s_1, s_2, \ldots, s_n$, choose the days on which you're going to reboot so to maximize the total amount of data you proces.

a) Give an example of an instance w/ the following properties

- There's a surplus of data
- optimal sol reboots sys. at least twice

day	1	2	3	4	5
x	12	9	8	3	9
s	8	4	3	2	1

reset on days 2 & 4 for optimal solution

OPT (sol) = 8 + 0 + 8 + 0 + 8 = 24

b) Give an efficient algorithm that takes values for $x_1, x_2, \ldots, x_n$ & $s_1, s_2, \ldots, s_n$ & returns total number of terabytes processed by optimal solution

Basic Idea: Given the input of $x_1, x_2, \ldots, x_n$ & $s_1, s_2 \ldots s_n$ we generate a container, v, of size $x_n + 2$ an initialize each value to zero. Then we iterate through $s_1, s_2 \ldots s_n$ and v[index] is equal to the max of $v[index-2] + \min(s[0], x[index])$ or $v[index-1] + \min(x[index], s[counter])$, where counter is initialized to zero prior to the loop, incremented by one each iteration, and if v[index] is equal to v[index-2] plus the minimum of (x[0], s[index]) then count = 0.

After the loop simply return the value of V at the index n-1. V[n-1] = max

Code: ★ these are meant to be arrays

```
def maxter(xlist, slist):
Initialize container → V = [0]*(len(xlist)+2)
    count = 0;
    for index in range(len(slist)):
        V[index] = max( V[index-2] + min(slist[0], xlist[index]),
            V[index-1] + min(xlist[index], slist[count]))
        if V[index] == V[index-2] + min(slist[0], xlist[index]):
            count = 0
        count += 1
    return V[len(xlist)-1]
```

Proof of correctness: This works because it simply iterates through each move and chooses the max value of either proceeding with the current sp value or rebooting and checks the max of these values two iterations over. In the array V, the value stored at each index is the best possible move, which is why returning the index n-1, where n is the size of array it returns the max terabytes.

Runtime: O(n) as the function simply uses 1 loop with additional array search and the can be done in O(1), so the function is simply iterating through $x_1, x_2 \ldots x_n$ & $s_1, s_2 \ldots s_n$ at the same time.

Space Complexity: O(n) as only an array of size xlist+2 is created, n=sizeof(xlist)

1 computing system 25 / 25

✓ **- 0 pts** **Correct**

- 2 pts minor mistake

- 3 pts a: not stating the optimal solution

- 20 pts b: Incorrect algorithm

- 10 pts b: Algorithm not optimal

- 10 pts b: major mistake

- 5 pts a) incorrect

- 5 pts b: not complete

2) Design an algorithm to find the minimum number of characters required to make a given string to a palindrome if you're allowed to insert chars at any position in the string.

Basic Idea: Let the input string be named str with index i to j. In order to solve this problem recursively, we will need to:

1. Find the minimum number of insertions in the substring str[i+1, ... j]
2. Find the minimum number of insertions in the substring str[i, ... j-1]
3. Find the minimum number of insertions in the substring str[i+1, ... j-1]

Recursively this can be done by:

```
if str[i] == str[j]:
    (minInsert(str[i+1 ... j-1])
else:
    min(minInsert(str[i ... j-1], minInsert(str[i+1, ... h]))+1
```

Now to use dynamic programming, we can re-use the subproblems we in the recursive case.

We can use the top down memoization technique to avoid the subproblem recalls. This can be done by creating a table-like data structure to store results of subproblems so that they can be used directly if the same subproblem is encountered again.

since we need a table of size^2, the space complexity is $O(n^2)$

Code:

```
int minInsert(char str[], int size):
    int table[size][size], i, j, buf;
    //make a table like structure & other vars
    memset(table, 0, sizeof(table));
    for(buf=1; buf<size; ++buf):
        for(i=0, j=buf; j<size; ++i, ++j):
table[i][j] = (str[i]==str[j])? table[i+1][j-1] :
            (min(table[i][j-1], table[i+1][j])+1);
    }
    }
    return table[0][size-1];
}
```

Initialize table to 0

fills table in order

returns min # of insertions from 0 → size-1

Proof: need to show recurrance relation

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

where str[i,j] represents the minimum insertions to make str a palindrome

Need to prove:

1. $str[i,j] \leq str[i+1, j-1]$
2. $str[i,j] \geq str[i+1, j-1]$

1. if $str[i+1], \ldots, str[j-1]$ can be converted into a palindrome using min chars, then it must be the case that $str[i], \ldots str[j]$ can be turned into a palindrome using at most min chars. min = $C[i+1, j-1]$
2. if $str[i], \ldots str[j]$ can be turned into a palindrome, then it must be the case that $S[i+1], \ldots, S[j-1]$ can be turned into a palindrome by inserting at most the same amount of chars.

Time complexity: $O(n^2)$ because of the nested for loop that iterates through n items n times with every other operation being less than $O(n^2)$. Space complex: $O(n^2)$

2 palindrome 25 / 25

✓ - **0 pts** **Correct algorithm**

- **25 pts** Incorrect algorithm
- **8 pts** Missing Psuedocode
- **5 pts** No explanation of runtime or incorrect runtime
- **5 pts** No explanation of correctness of algorithm

3) Given an undirected graph $G=(V,E)$, an independent set is a subset $I \subseteq V$ s.t. no two nodes in I are adjacent in G. Find a max cardinality independent set in a graph is a hard problem. Design an algorithm which, given a tree $T=(V,E)$ runs in $O(|V|)$ time & returns a max cardinality independent set in T.

Proof: Basic Idea: Root the tree at an arbitrary vertex. Now each vertex defines a subtree. Using dynamic programming, from smaller → larger subproblems, bottom up in the rooted tree. Suppose we know the size of the largest independent set of all subtrees below a vertex j. The max independent set is either j is in the max independent set or not

1. If it is not, max independent set is the union of the max independent sets of the subtree of the children of j.
2. If it is, then the max independent set consists of j plus the union of the max independent sets of the subtree of the grandchildren of j.

Let $I(j)$ be the size of the max independent set in the subtree rooted @ vertex j.

$$I(j) := \max\left(\sum_{k \text{ child of } j} I(k) \;,\; 1 + \sum_{k \text{ grandchild of } j} I(k) \right)$$

If v is a leaf, there exists a max size independent set containing v

proof of prev statement

Consider a max cardinality independent set S

If $v \in S$, ☒

If $v \notin S$ & $v \notin S$, then $S \cup \{v\}$ is independent $\Rightarrow$ S not max

if $v \in S$ & $v \notin S$, then $S \cup \{v\} - \{u\}$ is independent

Pseudo Code:

```
IndependentSet(tree T){
    S = {0}
    While ( T has atleast an edge){
        let e = (u,v) be an edge such
that v is a leaf
        add v to S
        delete from T vertex u & v, &
all edges incident to them
    }
    return S
}
```

Time complexity: $O(|V|)$ as the algorithm only looks at the children & grandchildren of each vertex, so each vertex is only look at at most 3 times (constant time), the runtime is $O(|V|)$

Space complexity: $O(n)$ as we only need a set that will atmost be the # of vertices

3 IS **25 / 25**

✓ **- 0 pts** **Correct**

- 25 pts incorrect

- 10 pts major mistake

- 15 pts Solution incomplete

- 5 pts minor mistake

4) Consider the set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots B_m$ of subsets of A. We say that a set $H \subseteq A$ is a hitting set for the collection $B_1, B_2, \ldots B_m$ if H contains at least one element from each B_i - that is, if $H \cap B_i$ is not empty for each i. We now define the Hitting Set Problem as follows: We are given a set $A = \{a_1, \ldots, a_n\}$ a collection $B_1, B_2, \ldots, B_m$ of subsets of A, & a #K. Is there a hitting set $H \subseteq A$ for $B_1, \ldots, B_m$ so that the size of H is at most K. Prove that the vertex cover problem $\leq_p$ the hitting set problem.

In order to show that hitting set is NP, the solution just needs to exhibit the set H as one can easily verify that in polynomial time whether H is of size K & intersects each of the sets $B_1, \ldots, B_m$. We reduce from the vertex cover problem. Consider an instance of the vertex cover problem where graph $G = (V, E)$ w/ a size track integer K. We map this to an instance of the hitting set problem as follows. The set A is the set of vertices V and for every edge $e \in E$, we have a set S_e consisting of the 2 endpoints of edge e. It is easy to see that a set of vertices S is a vertex cover of G if and only if the corresponding elements form a hitting set in the hitting set instance.

In order to prove the hitting set is NP-Complete we must 1st prove it's in NP. The checking algorithm will verify whether it's actually a hitting set by checking the "∩" of H. B $B_1, B_2, \ldots B_m$ which will take time polynomial m.
This proves the hitting set is in NP. Now, we must prove the hitting set is NP-Complete.
Let's say we have some NP-complete problem P_1. Now, we have to prove $P_1 \leq_p$ hitting set problem. Let's consider P_1 to be the vertex cover problem. Now if we prove the vertex cover $\leq_p$ hitting set problem, this implies that the hitting set problem is NP-complete. We can this by transforming an instance of the VC Prob. Given a graph $G=(V,E)$ is there's a vertex of size $\geq K$ to an instance of the hitting set. Given set $A=\{a_1, \ldots, a_n\}$ & subset $B_1, B_2, \ldots B_m$ w/ an integer K' a hitting size @ most K.
For each edge (u,v) in E add a subset $B_i=\{u,v\}$ which is equivalent to a subset for each edge in G
$K''=K$
If there's a vertex cover in G of size $\geq K$ iff there's a hitting set of size $\geq K'$
If there's a vertex cover C of size $\geq K$, by def $\forall (u,v)$ in E either u and/or v in C, then $\forall B_i$ $C \cap B_i \neq \emptyset$, then C is also a hitting set of size at most K. This allows us to prove that the transformation is polynomial by analyzing:
- $A=V$: takes $O(n)$ time
- For each edge (u,v) in E add a subset $B_i=(u,v)$ takes $O(m)$ time
- $K' \leq K$ constant time

vertex cover $\leq_p$ hitting set problem
$\therefore$ is NP-comp

4 Hitting set **10 / 10**

✓ **- 0 pts** **Correct**

- 10 pts Incorrect

- 5 pts proof not clear enough

5) An undirected graph $G=(V,E)$ is called "K-colorable" if there exists a way to color the nodes w/ K colors s.t. no pair of adjacent nodes are assigned the same color. Prove that the 3-colorable problem $\leq_p$ the 4-colorable problem

Using a verifier for the 4-colorable problem, showing the latter is NP

verifier $V(\langle G\rangle, \langle c\rangle)$ where $G=(V,E)$
Accept if and only if
- c is a map from V to $\{1,2,3,4\}$
- $\chi(u) \neq \chi(v)$ for all $u,v \in E$

Now, in order to show 3-colorable problem $\leq_p$ 4-colorable problem
The reduction function f takes input graph $G=(V,E)$ & produces graph $G'=(V',E')$.
Introduce a vertex w that is not in V and let $V' = V \cup \{w\}$. Let $E' = E \cup \{\{v,w\} : v \in V\}$
If $c: V \Rightarrow \{1,2,3\}$ is a 3 colorable problem of G, then define $c': V' \Rightarrow \{1,2,3,4\}$ by $c'(v) = c(v)$ for $v \in V$ & $c'(w) = 4$.
This is a 4 colorable problem of G'. WLOG, the color assigned to w is 4 but since w is connected to all vertices in V, no vertex in V has $c(v)=4$. Thus, the map $c: V \rightarrow \{1,2,3\}$ defined by $c(v) = c'(v)$ for all $v \in V$ is a 3 colorable of G. This proves that if G' is a 4 colorable, then G is a 3 colorable.

5 K-coloarable 10 / 15

- **0 pts** Correct
- **10 pts** 1. Did not prove that G is 3-colorable if and only if G' is 4-colorable.
- ✓ **- 5 pts** **2. Did not prove that G' can be computed from G in polynomial time.**