CIS 675 (Fall 2018) Disclosure Sheet

Name: Wentan Bai
HW # 5

Yes **No** Did you consult with anyone on parts of this assignment, including other students, TAs, or the instructor?

Yes **No** Did you consult an outside source (such as an Internet forum or a book other than the course textbook) on parts of this assignment?

If you answered **Yes** to one or more questions, please give the details here:

I also consult all questions and extra question with another student, Wentian Bai. For all questions, we discussed our ideas and I finish my algorithms independently.

By submitting this sheet through my Blackboard account, I assert that the information on this sheet is true.

This disclosure sheet was based on one originally designed by Profs. Royer and Older.

Question 1:

Suppose the given grid has n numbers of rows and m numbers of columns.

- On the Westernmost road, all T[i][1] = T[i+1][1] + $d_{i,1}$, since we can only go straight north to the current position at the previous intersection.
- On the Southernmost road, all T[n][j] = T[n][j-1] + $d_{n,j}$, since we can only go straight east to the current position at the previous intersection.
- Then for remaining intersections, we can go straight east or straigth north to the current position at the previous intersections. Therefore, we choose the shorter one, T[i][j] = $d_{i,j}$ + **min**(T[i+1][j], T[i][j-1])

T[n][m]
T[n][1] = $d_{n,1}$
For i ← n-1 to 1 **:**
    T[i][1] ← T[i+1][1] + $d_{i,1}$
For j ← 2 to m **:**
    T[n][j] ← T[n][j-1] + $d_{n,j}$

For i ← n-1 to 1 **:**
    For j ← 2 to m **:**
        T[i][j] = $d_{i,j}$ + **min**(T[i+1][j], T[i][j-1])

Return T[1][m]

---

running time:

We iterate through all intersections so the final time is $O(n^2)$.

Question 2:

This question is similar as Splitting a String. I need to write a helper function to determine whether a particular string is a palindrome.

- we check if a sub string from i to j is a palindrome. If current sub string is a palidrome, we check if the previous sub string is a palidrome, which result is saved at Seq(i-j-1). If all requirements are satisified, save Seq(i) as True.
- The helper function is to determine whether a particular string is a palindrome. We iterate from head and tail to center at the same time. If there are a pair of characters which are different, this particular string is not a palindrome.

```
Seq(i) = False
Seq(0) = True
For i = 1 to n :
    For j = 1 to i :
        if helper(Seq[i-j ... i]) == True and Seq(i-j-1) == True
            Seq(i) ← True
Return Seq(n)
```

Following is helper function:

```
def helper(Seq)
    if length(Seq) == 1 or 0
        Return False
    i = 0, j = length(Seq) - 1
    While i ≠ j and i < j :
        if Seq[i] ≠ Seq[j] :
            Return False
        i++, j- -
    Return True
```

---

running time:

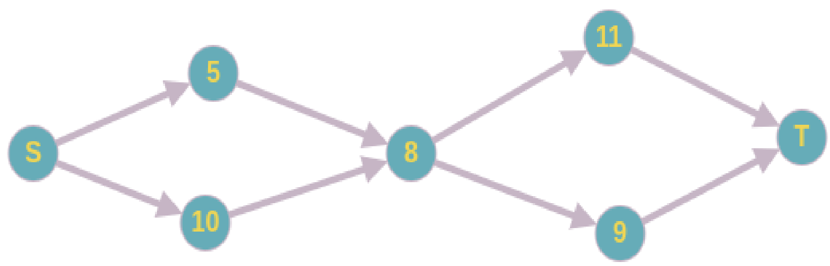
When we iterate through all sub strings, at worst case we need $O(n^2)$. The final time is $O(n^2)$

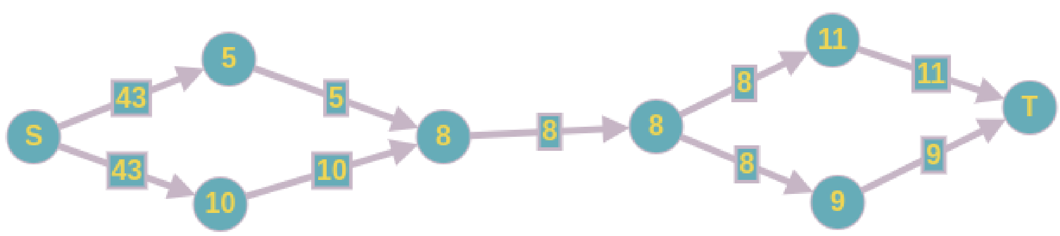
Question 3:

By hint, I modify the graph. In original graph, there are not capacities on the edges, so I will add capacities for each edges.

- Assign ∞ or the sum of capacities of all nodes to the capacities of all edges leaving from source node.
- If the current node is only pointed by one node and only one edge leaves from this node. Then assign the same size of capacity on current node to the edge leaving from current node.
- If the current node u points to multiple nodes or is pointed by multiple nodes, then make node u only connect with a new node v which has the same capacity. Then assign the same size of capacity on node u to the $Edge(u, v)$ Connect node v to the nodes which are original pointed by node u. Assign the same size of capacity on node u to the capacities of these new edges.

Following is an example. The first one is original graph and the second is modified graph.





We can find a max flow using original maximum flow algorithm on the modified graph.

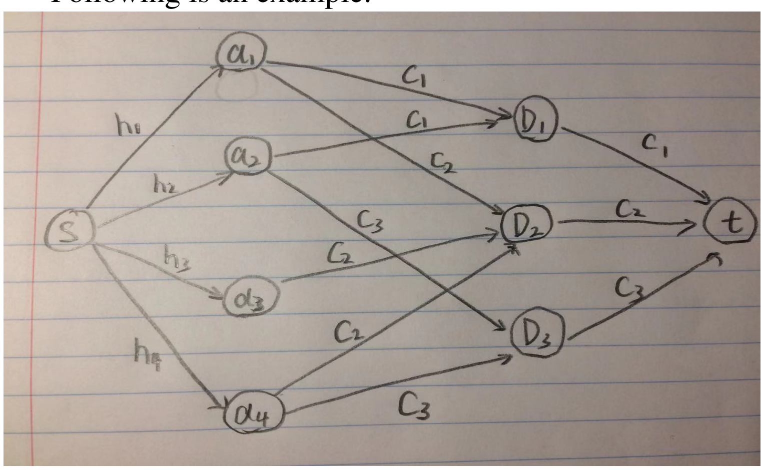
running time:
The time is the same as the time of original maximum flow algorithm, $O(|V| \cdot |E|^2)$.

Question 4:

My algorithm is:

- Create a node for each animal denoted as a_i. Then create a node for each doctor denoted as D_j. Create a source node s and a sink node t.
- Draw edges from source node to every node a_i, and each $Edge(s, a_i)$ has a capacity h_i. Input flow is the time of the animals to be treated.
- If animal a_i can be treated by doctor D_j, draw a edge from a_i to D_j, and each $Edge(a_i, D_j)$ has a capacity C_j.
- Draw edges from every doctor D_j to sink node t, and each $Edge(D_j, t)$ has a capacity C_j. This setp ensures each doctor j works at most C_j time.
- Run maximum flow algorithm on this graph.

Following is an example.



We can get the answer by using original maximum flow algorithm on this graph.

running time:

The time is the same as the time of original maximum flow algorithm, $O(|V| \cdot |E|^2)$.

Question 5:

My algorithm is:

- Create a node for each paper denoted as p_j, and create a node for each reviewer denoted as R_i. Create a source node s and a sink node t.
- Draw edges from source node to every node p_j, and each $Edge(s, p_j)$ has a capacity 3.
- Draw a edge from each node p_j to every node R_i if paper p_j is not submitted by reviewer R_i. Each $Edge(p_j, R_i)$ has a capcity 1.
- Draw edges from every node R_i to sink node, and each $Edge(s, p_i)$ has a capacity m_i.
- Run maximum flow algorithm. However, in this algorithm, we must ensure that in the final graph all flows from source node to each node p_j is 3, which means that each node p_j is reviewed by 3 times. The final flow into sink node should be 3 times of numbers of papers.

running time:

The time is the same as the time of original maximum flow algorithm algorithm, $O(|V| \cdot |E|^2)$.

Extra:

Create an array $dp[M][N]$: $dp[i][j]$ presents we need to choose i numbers of fruits from j numbers of boxes.

In initialization, if $m_1 \geq 1$, assign $dp[1][1]$ to 1; else assign $dp[1][1]$ to 0.

For all $dp[1][2]$ to $dp[1][N]$, in each step j, if $m_j \geq 1$, then assign $dp[1][j]$ = $dp[1][j-1]+1$; else assign $dp[1][j] = dp[1][j-1]$. In each step j, we need to choose one fruit from j numbers of boxes. If B_j is not empty, then there will be $dp[1][j-1]+1$ ways to choose. Otherwise, there are still the $dp[1][j-1]$ numbers of ways.

For all $dp[2][1]$ to $dp[M][1]$, in each step i, if $m_1 \geq i$, then assign $dp[i][1] = 1$; else assign $dp[i][1] = 0$. In each step i, we need to choose i numbers of fuits from B_1. If m_1 is larger than required numbers of fruits i, then there is one type to choose; Otherwise, there is not enough numbers of fruits to choose.

Iterate from $i = 2$ to M, and for each i, iterate from $j = 2$ to N, and in each step i j, assign $dp[i][j] = dp[i-1][j] + dp[i][j-1]$.

The final result is in $dp[M][N]$.

---

dp[M][N]
dp[1][1] $\leftarrow$ 0

For j = 2 to N :
  If $m_j \geq 1$ **:**
    $dp[1][j] \leftarrow dp[1][j-1]+1$
  Else :
    $dp[1][j] \leftarrow dp[1][j-1]$

For i = 2 to M :
  If $m_1 \geq i$ **:**
    $dp[i][1] \leftarrow 1$
  Else :
    $dp[i][1] \leftarrow 0$

For i = 2 to M :
  For j = 2 to N :
    $dp[i][j] \leftarrow dp[i-1][j] + dp[i][j-1]$

return $dp[M][N]$

---

running time:

Interation from 2 to M using M time, and for each step, interation from 2 to N using N time. Therefore, the final time is $O(NM)$