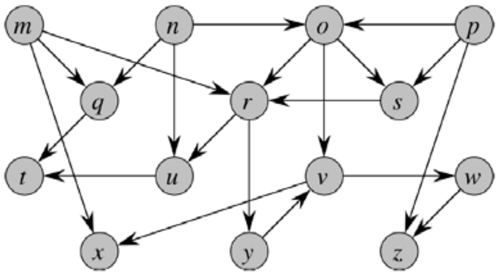
EL9343 Homework 5

(Due Nov 27th, 2020)

No late assignments accepted

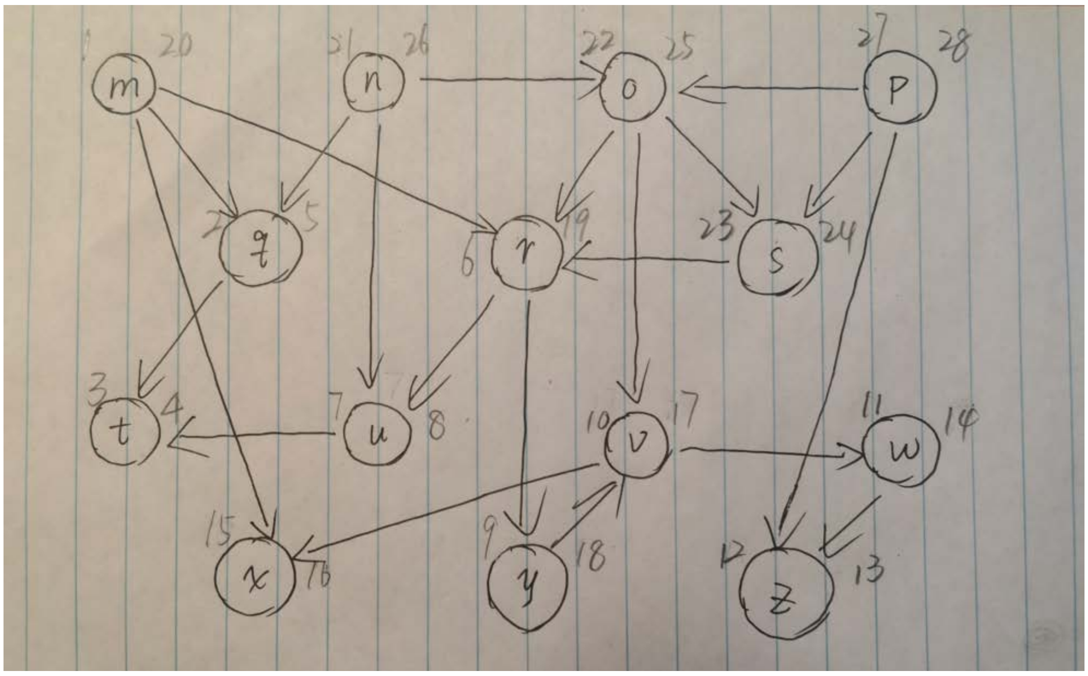
All problem/exercise numbers are for the third edition of CLRS text book

1. Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the DAG below. Assume that **for** loop of lines 5—7 of the DFS procedure (page 604 in CLRS) considers the vertices in alphabetical order, and assume the adjacency list is ordered alphabetically.



Solution:

This is the graph after we run DFS on the graph:



The left number is the discover time and the right number is the finish time.

According to the TOPOLOGICAL-SORT(G), we can get the ordering of the vertices should be:

p, n, o, s, m, r, y, v, x, w, z, u, q, t.

2. Run the procedure STRONGLY-CONNECTED-COMPONENTS on the graph below. Show the:

   (a) The finishing times for each node after running DFS in line 1

   (b) The DFS forest produced by line 3

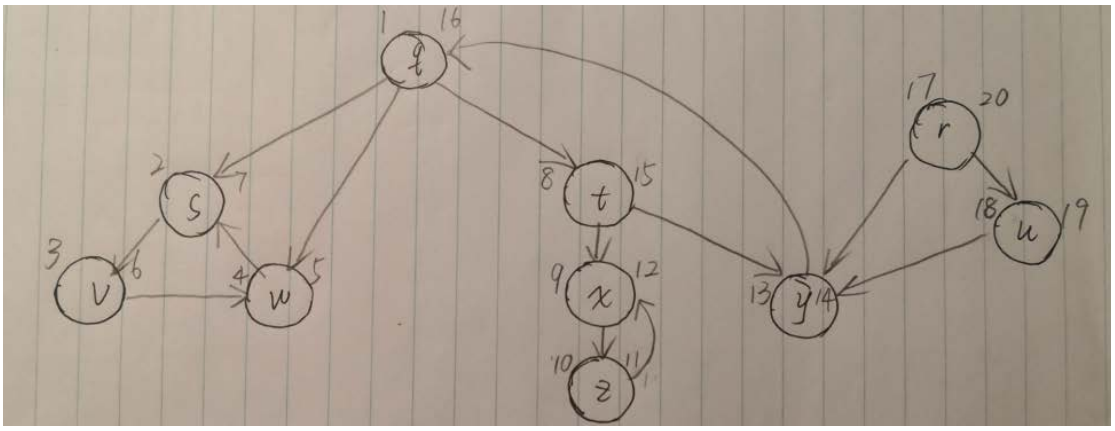
   (c) The nodes of each tree in the DFS forest produced in line 3 as a separate strongly connected component.

   Assume that **for** loop of lines 5—7 of the DFS procedure (page 604 in CLRS) considers the vertices in alphabetical order, and assume the adjacency list is ordered alphabetically.

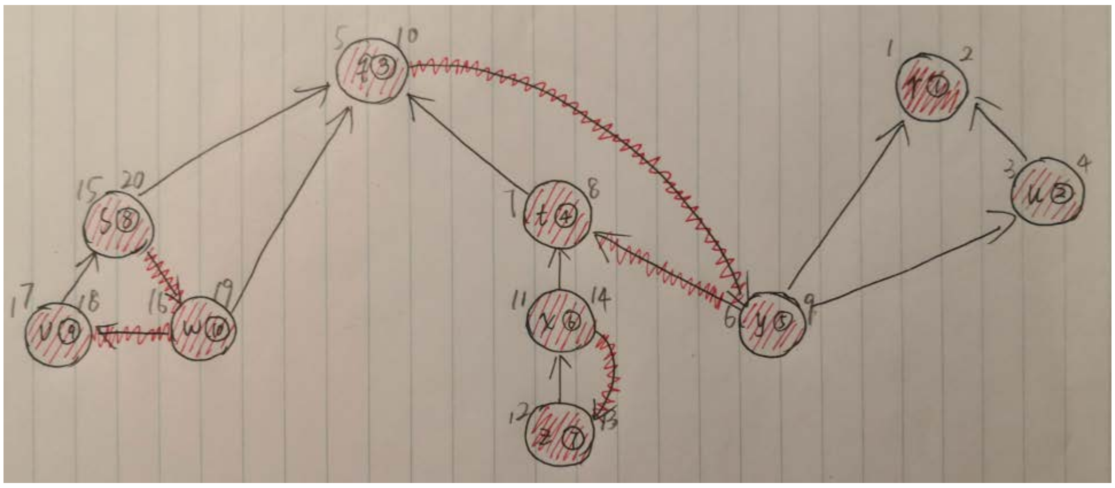
Solution:

(a)

As shown below, the discover time and finish time lie on two sides of each node, the left is the discover time, the right is the finish time.



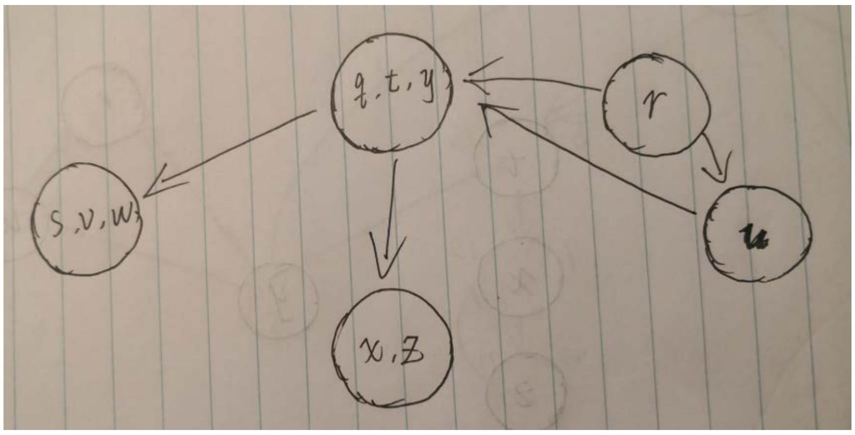
(b)



The result of DFS on G^T is shown as above. The number inside each node is the order computed from (a). Discover time and finish time lie on two sides of each node, the left is the discover time, the right is the finish time. Every DFS tree is marked by red pen. As we can see, there are 5 distinct trees. Which means the G has 5 strongly connected component.

(c)

Converge the nodes in one tree in (b) to one node and link them:



3. Define the edit distance between two string *X* and *Y* of length *m* and *n*, respectively, to be the number of edits that it takes to change *X* into *Y*. An edit consists of a character insertion, a character deletion, or a character replacement. For example, the strings "algorithm" and "rhythm" have edit distance 6. Design an *O*(*mn*)-time algorithm for computing the edit distance between *X* and *Y*.

Solution:

If we have 2 strings X_i, Y_j. If $x_i = y_j$, we can just edit X_{i-1} and Y_{j-1}. That is, we will ignore the last letter of X_i and Y_j. And we will not compare the edit distance of X_{i-1} and Y_{j-1} with the edit distance of X_{i-1} and Y_j or edit distance of X_i and Y_{j-1}. The reason is that the edit distance of X_{i-1} and Y_{j-1} is at most one more than edit distance of X_{i-1} and Y_j or the edit distance of X_i and Y_{j-1} (Which can be proved by the LCS). Else if $x_i \neq y_j$, we can modify one of them and then edit the rest, X_{i-1} and Y_{j-1}. When $x_i \neq y_j$, we have several edition method, we only analyze the edition of the first string, because considering the other string is symmetric:

(1) Replace x_i with y_j. Now, we only need to find the edit distance between X_{i-1} and Y_{j-1}.

(2) Add a y_j at the back of X_i. Now, we only need to find the edit distance between X_i and Y_{j-1}.

(3) Remove x_i. Now, we only need to find the edit distance between X_{i-1} and Y_j.

Obviously, any one above including an edition of one letter (x_i) and the edition of the rest 2 strings, we need to find the minimum of them to get our minimum edit distance of X_i and Y_j.

Let m is the length of X, n is the length of Y, $d[0..m][0..n]$ is a matrix, $d[i][j]$ is the edit distance of X_i and Y_j, we can easily get the following recurrence formula:

$d[i][j] = d[i-1][j-1]$ If $x_i = y_j$.

$d[i][j] = \min(d[i-1][j-1]+1, d[i][j-1]+1, d[i-1][j]+1)$ If $x_i \neq y_j$

Now, we can get the algorithm according to the formula above:

```
EDIT_DIS($X$, $Y$)    // $X$, $Y$ are strings
    $m = X$.length
    $n = Y$.length
    Let $d[0..m][0..n]$ be new array
    for $j$ form $0$ to $n$
        $d[0][j] = j$
    for $i$ form $1$ to $m$
        $d[i][0] = i$
    for $i$ from $1$ to $m$
        for $j$ from $1$ to $n$
            if $X[i] == Y[j]$
                $d[i][j] = d[i-1][j-1]$
            else  // $X[i] \neq Y[j]$
                $d[i][j] = \min(d[i-1][j-1]+1, d[i][j-1]+1, d[i-1][j]+1)$
    return $d[m][n]$
```

Because we compute the value of a matrix with dimension $(m+1)\times(n+1)$, and computing every element can be completed in constant time, the total running time is $O(mn)$. (Of course, we can derive it from the code, one loop with time cost $O(n)$, one loop with time cost $O(m)$ and a loop having an inner loop with time cost $O(mn)$, the total running time is also $O(mn)$).

4. The holidays are here! You decide to travel to see your friends in California.

   Since you are on a budget, so you decide to drive to visit them. You choose your route so you pass by the most spectacular vistas on your route. Along your route are n hotels, at miles $m_1 < m_2 < \cdots < m_n$ from your starting location. The cost of the i^{th} hotel is h_i. Your friends live at mile marker m_{n+1} where $m_n < m_{n+1}$. The starting location is m_0.

You won't drive more than 400 miles per day for safety concerns. (The hotels on your route are all within 400 miles of each other)

You want to minimize the cost of your travel.

Provide a recurrence formula that would calculate the answer.

Using your recurrence formula, design an algorithm to find the minimum cost.

Solution:

Let $c[0..n]$ is an array. $c[i]$ denotes that the maximal total resource consumption when the last rest points of the whole venture is the i^{th} rest point. Once he arrives this point, his total resource consumption is determined. So, $c[i]$ only depends on those rest points are before the i^{th} point, but the distance of every one of them must be larger than or equal to p miles to the i^{th} point. We can have the following formula: Obviously, $c[0] = 0$. For every $0 < i \le n$, we have:

$c[i] = \max_{0 \le j < i}(c[j] + r_i)$ When $d_i - d_j \ge p$.

After we compute every $c[i]$, we need to compare all of them and pick the largest one.

So, we can get our algorithm directly from the formula above.

```
MAX_CONSUM($n$, $d[0..n]$, $r[1..n]$)
  Let $c[0..n]$ be new array
  $c[0] = 0$
  for i from 1 to n
      temp = $-\infty$
      for j from 0 to i - 1    // We want to find the first one that is less than $p$ miles away from ith point and then no need to find those points which is behind that point.
          if $d[i] - d[j] \ge p$    // larger than $p$ miles
              if $c[j] + r[i] >$ temp
                  temp = $c[j] + r[i]$
          else // within $p$ miles, we do not need to consider these points
              break
```

```
    max_consum = 0
    for i from 1 to n
        if $c[i] >$ max_consum
            max_consum = $c[i]$
    return max_consum
```

Because we have an outer loop and an inner loop, and all operations in a loop is in $O(1)$ time. Running time for it is $O(n^2)$. After we compute every element in $c[0..n]$, we need a loop to pick the largest one, which is in $O(n)$ time. Totally we will need $O(n^2)$ time to complete this task.

5. A native Australian named Alice wishes to cross a desert carrying only a single water bottle. She has a map that marks all the watering holes along the way. Assuming she can walk k miles on one bottle of water, design an efficient algorithm for determining where Alice should refill her bottle in order to make as few stops as possible. Argue why your algorithm is correct. Justify the running time of your algorithm.

Solution:

We just decide to refill our bottle at the hole that is the last one within the k miles of the hole we refill last time, and so force we will get to the destination. This refill method will give the solution with fewest stops. Let $S[0..n+1]$ be an array, $S[i]$ ($i = 1,$ 2, …, n) records the distance of the ith watering hole from the start point (the 1st watering hole). $S[i] = 0$, represent the distance of the start point, $S[n+1]$ is the distance of the destination. We can get the following pseudocode:

```
FEWEST_STOP($S[0..n+1]$)
    num_refill = 0
    current_position = 0
    last_refill_position = 0
    while current_position < n + 1
        last_refill_position = current_position
        while (current_position < n + 1 and
                S[current_position + 1] - S[last_refill_position] <= k)
            current_position = current_position + 1
        if current_position < n + 1
```

```
            num_refill += 1
    return num_refill
```

The running time of this algorithm is $O(n)$, because current_position is increasing all the time and changes 1 by 1, and outer loop restrict it only can reach n + 1. For every current_position, our operation can be done in $O(1)$ time. So, the total running time is $(n+1)O(1) = O(n)$. Now we prove the correctness of this algorithm:

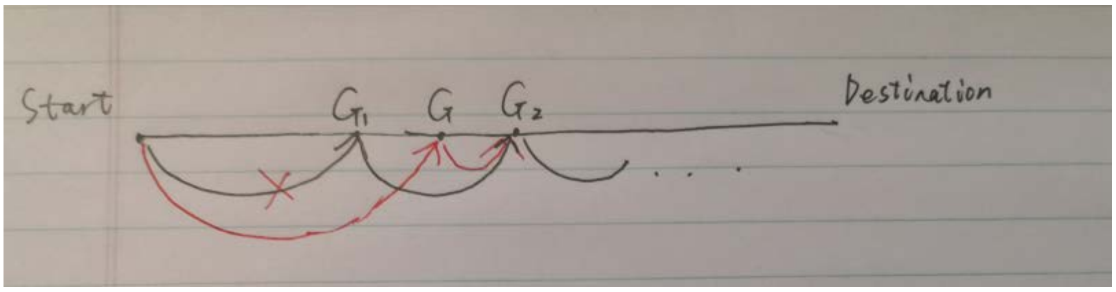
(1) Optimal Substructure:

Every time we arrive at some water hole i, the minimal number of refill time is the smallest one among all possible last stops: The last stop can be at 1st hole, 2nd hole, …, (i-1)th hole. So, this problem has optimal substructure.

(2) Greedy Choice Property:

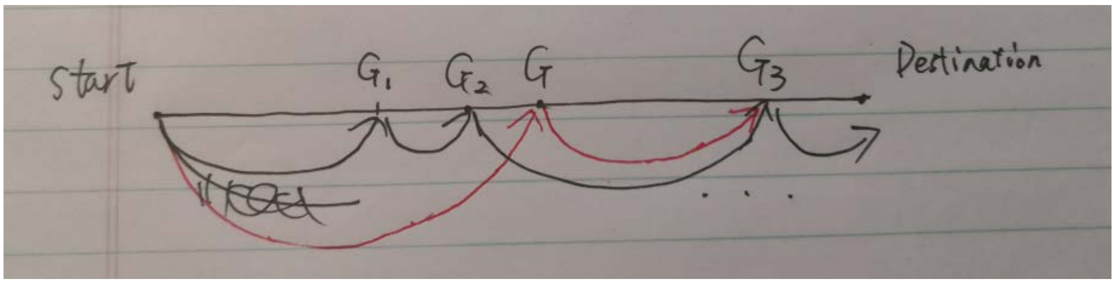
We can prove that our choice of refilling in every step will make up of the optimal solution. Let G be the hole that is the last one within k miles from our start point. If we have an optimal solution that has the fewest stops along the way and the stops are G_1, G_2, …, G_T (Assume that the fewest stops is T), and $G_1 \neq G$, which means G_1 is before G, we will have the following cases:

(a) G_2 is behind G



As we can see, the path consist with black directed edges is our optimal route, we can replace the first 2 black edges with 2 red edges in the picture above, the 2 red edges and the following black edges will also be an optimal route of our problem.

(b) G_2 is before G



As we can see, the path consist with black directed edges is our optimal route. If we replace the first 3 black edges with 2 red edges, we can get a path that has $T-1$ stops, it contradicts with the hypothesis that our stops are $G_1,\ G_2,\ \ldots,\ G_T$ is an optimal solution. This case is impossible.

So, we only have case (a), we have proved that every time choosing the hole that is the last one within k miles of our last refill hole as our refill hole is safe. Which means this problem has greedy choice property, the algorithm above will work.

6. Suppose you've been sent back in time and have arrived at the scene of an ancient Roman battle. Moreover, suppose you have just learned that it is your job to assign n spears to n Roman soldiers, so that each man has a spear. You observe that the men and spears are of various heights, and you have been told that the army is at its best if you can minimize the total difference in heights between each man and his spear. That is, if the i^{th} man has height m_i and his spear has height s_i, then you want to minimize the

$$\sum_{i}^{n} |m_i - s_i|$$

Consider a greedy strategy of repeatedly matching the man and the spear that minimizes the difference in heights between these two. Prove or disprove that this greedy strategy results in the optimal assignment of spears to men.

Solution:

Let $m[1..n]$ and $s[1..n]$ be arrays, $m[i]$ represents the height of ith man and $s[i]$ represents the height of the ith spear. We firstly sort them and just assign $s[i]$ to $m[i]$. Then the sum $\sum_i^n |m[i]-s[i]|$ will be minimized. Pseudocode is shown as below:

```
OPT_DIF_SUM($m[1..n]$, $s[1..n]$)
    Merge_Sort($m[1..n]$)
    Merge_Sort($s[1..n]$)
    sum = 0
    for i from 1 to n
        sum += abs($m[i] - s[i]$) //abs(a, b) will return the absolute value of a – b
    return sum
```

Every merge sort will cost $O(n \log n)$ time, and computing sum will cost $O(n)$ time because it is a loop with i form 1 to n. The total running time will be $O(n \log n) + O(n) = O(n \log n)$.

Now, we prove the correctness of this algorithm:

(1) Optimal Substructure:

When we split $m[1..n]$ and $s[1..n]$ to 2 part, $m[1..k]$, $m[k+1..n]$ and $s[1..k]$, $s[k+1..n]$, respectively ($k = 1,2,\dots,n$), the optimal assigning of $m[1..n]$ to $s[1..n]$ must be combined with the optimal assigning of $m[1..k]$ to $s[1..k]$ and the optimal assigning of $m[k+1..n]$ to $s[k+1..n]$. So, this problem has optimal substructure.

(2) Greedy Choice Property:

If we have an optimal solution that $s[n]$ is not assigned to $m[n]$, WLOG, assume that $s[n]$ is assigned to $m[u]$, and there must be a $s[v]$ is assigned to $m[n]$. Now, we have $s[n] \geq s[v]$ and $m[n] \geq m[u]$, because we sort $m[1..n]$ and $s[1..n]$. If we assign $s[n]$ to $m[n]$ and assign $s[v]$ to $m[u]$. We can also get an optimal solution:

$|s[n] - m[u]| + |s[v] - m[n]| \geq |s[n] - m[n]| + |s[v] - m[u]|$. In order to prove this, we can change some notation to conveniently represent our variables: When $a < b$, $c < d$, $|a - d| + |b - c| \geq |a - c| + |b - d|$.

(a) $a < b < c < d$

$|a - d| + |b - c| = d - a + c - b = -a - b + c + d$

$|a - c| + |b - d| = c - a + d - b = -a - b + c + d$

$|a - d| + |b - c| \geq |a - c| + |b - d|$ holds.

(b) $c < d < a < b$

$|a - d| + |b - c| = a - d + b - c = a + b - c - d$

$|a - c| + |b - d| = a - c + b - c = a + b - c - d$

$|a-d|+|b-c| \geq |a-c|+|b-d|$ holds.

(c) $a < c < d < b$

$|a-d|+|b-c| = d-a+b-c = -a+b-c+d$

$|a-c|+|b-d| = c-a+b-d = -a+b+c-d < -a+b < -a+b-c+d$

$|a-d|+|b-c| \geq |a-c|+|b-d|$ holds.

(d) $c < a < b < d$

$|a-d|+|b-c| = d-a+b-c = -a+b-c+d$

$|a-c|+|b-d| = a-c+d-b = a-b-c+d < -c+d < -a+b-c+d$

$|a-d|+|b-c| \geq |a-c|+|b-d|$ holds.

(e) $a < c < b < d$

$|a-d|+|b-c| = d-a+b-c = -a+b-c+d$

$|a-c|+|b-d| = c-a+d-b = -a-b+c+d < -a+d < -a+b-c+d$

$|a-d|+|b-c| \geq |a-c|+|b-d|$ holds.

(f) $c < a < d < b$

$|a-d|+|b-c| = d-a+b-c = -a+b-c+d$

$|a-c|+|b-d| = a-c+b-d = a+b-c-d < b-c < -a+b-c+d$

$|a-d|+|b-c| \geq |a-c|+|b-d|$ holds.

As we can see, in any case, if $a < b$, $c < d$, $|a-c|+|b-d| \leq |a-d|+|b-c|$. In other case, $s[v] < s[n]$, $m[u] < m[n]$. So, $|s[v]-m[u]|+|s[n]-m[n]| \leq |s[n]-m[u]|+|s[v]-m[n]|$. We can see, if there is an optimal solution, we can assign $s[n]$ to $m[n]$ to make a solution that is also the optimal one. So, this problem has greedy choice property. It is safe for us to do the greedy algorithm above.