CS570
Analysis of Algorithms
Fall 2015

Exam II

Name: ______________________

Student ID: __________________

Email Address:________________

_____Check if DEN Student

	Maximum	Received
Problem 1	20	
Problem 2	20	
Problem 3	20	
Problem 4	20	
Problem 5	20	
Total	100	

Instructions:
1. This is a 2-hr exam. Closed book and notes
2. If a description to an algorithm or a proof is required please limit your description or proof to within 150 words, preferably not exceeding the space allotted for that question.
3. No space other than the pages in the exam booklet will be scanned for grading.
4. If you require an additional page for a question, you can use the extra page provided within this booklet. However please indicate clearly that you are continuing the solution on the additional page.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[TRUE]

The Ford-Fulkerson Algorithm finds a maximum flow of a unit-capacity flow network with *n* vertices and *m* edges in time *O(mn).*

[/FALSE]

In a flow network, if maximum flow is unique then min cut must also be unique.

[/FALSE]

In a flow network, if min cut is unique then maximum flow must also be unique.

[/FALSE]

In dynamic programming you must calculate the optimal value of a sub-problem twice, once during the bottom up pass and once during the top down pass.

[TRUE/]

Bellman-Ford algorithm solves the shortest path problem in graphs with negative cost edges in polynomial time.

[TRUE/]

The problem of deciding whether a given flow f of a given flow network G is a maximum flow can be solved in linear time.

[/FALSE]

An optimal solution to a 0/1 knapsack problem will always contain the object *i* with the greatest value-to-cost ratio V_i/C_i

[TRUE/]

The Ford-Fulkerson algorithm is based on greedy.

[TRUE/]

A flow network with unique edge capacities may have several min cuts.

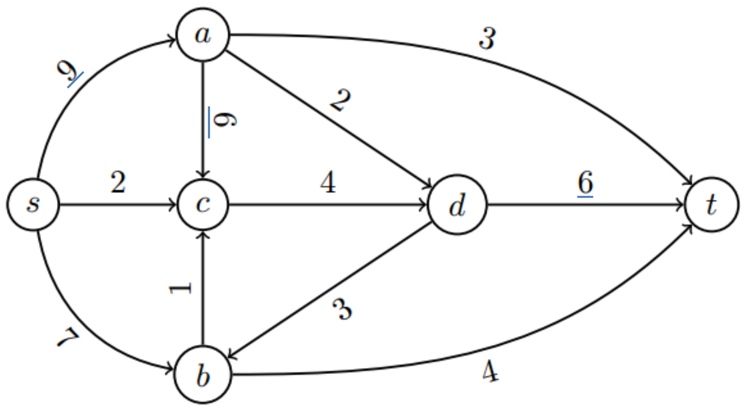
[/FALSE]

Complexity of a dynamic programming algorithm is equal to the number of unique sub-problems in the solution space.

2) 20 pts

Consider the flow network G below with source s and sink t. The edge capacities are the numbers given near each edge.

(a) Find a maximum flow in this network using the Ford Fulkerson algorithm. Show all steps of augmentation. Once you have found a maximum flow, draw a copy of the original network G and clearly indicate the flow on each edge of G in your maximum flow. (12 pts)



(b) Find a minimum s-t cut in the network, i.e. name the two (nonempty) sets of vertices that define a minimum cut. (8 pts)

Solution:

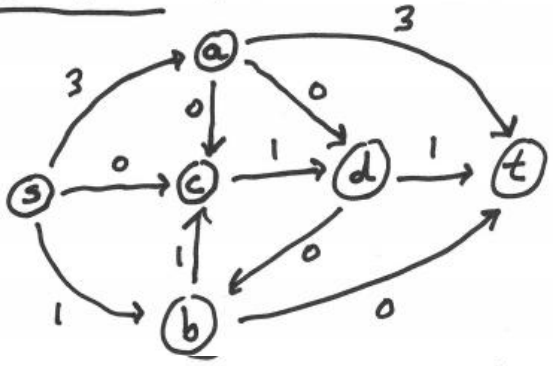
Starting from 0 flow (i.e. flow of 0 on each edge), we find augmenting paths.

We can find some that are non-overlapping and augment along all simultaneously.

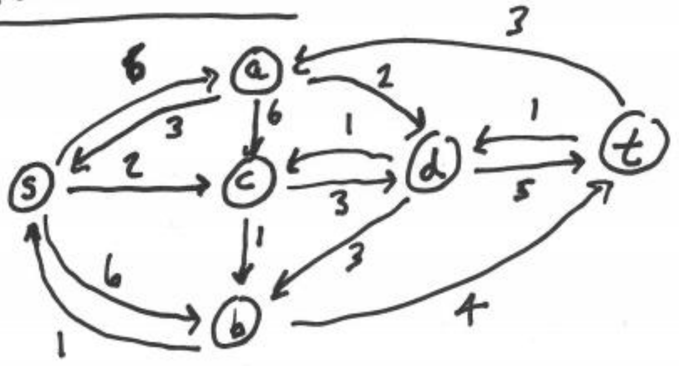
For example, we can take

s, a, t   augment 3 units
s, b, c, d, t   augment 1 unit

New flow (flows on edges)



Residual network (residual capacities on edges)

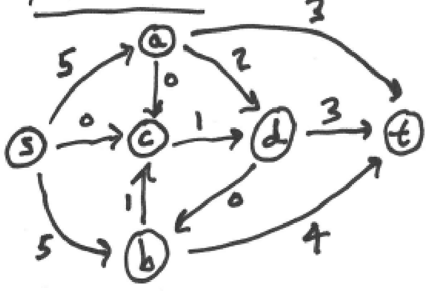


Augment flow on

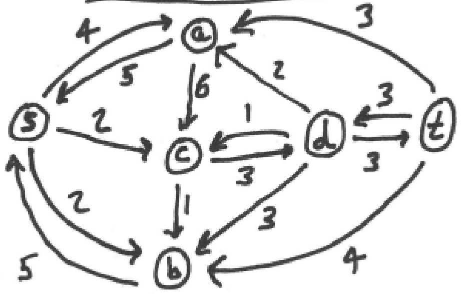
s, a, d, t 2 units

s, b, t 4 units

New flow

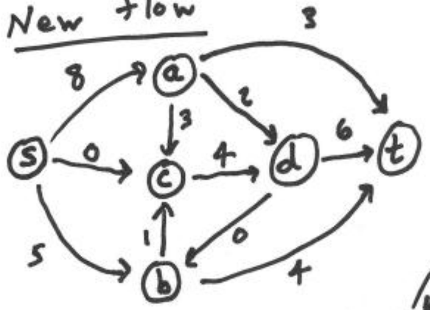


New residual network

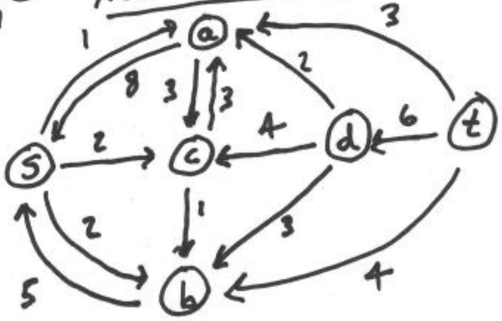


We can augment flow on the path s, a, c, d, t by 3 units.

New flow



New residual network



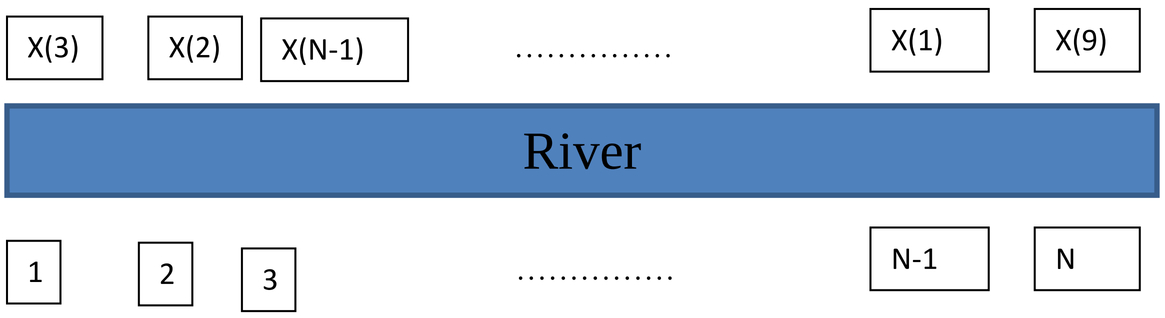
No more augmenting paths.

Thus, the max flow has value 13. Note that the flow shown is not unique, i.e. there are other possible maximum flows in the network, but they will all have the same value of 13.

(b)

As in problem (1), start from *s* and find which vertices can be reached in the final residual network for one set of the cut, and the remaining vertices form the other set. So one minimum cut is *{s, a, b, c} {d, t}*. The (forwards) edges across this cut are (*a*, *d*), (*a*, *t*), (*c*, *d*), and (*b*, *t*). The capacities of those edges are, respectively, 2, 3, 4, and 4, the sum of which is 13 (which it should be according to the Max Flow/Min Cut Theorem). Note, also in this case that the minimum cut isn't unique. For example, the cut consisting of the two sets *{s, a, b, c, d}* and *{t}* also has capacity 13.

3) 20 pts
There is a river going from east to west, on the south bank of the river there are a set of cities 1…N as shown in the figure below. On the northern bank there are also N cities. Each city on the south bank has a unique sister city on the north bank of the river. We use X(i) to denote the sister city corresponding to city i. Suppose you are assigned a task to build bridges to connect the southern cities with their sister cities such that as many bridges as possible are built without any bridges crossing each other. Solve this problem using dynamic programming.



a) Recurrence formula (7 pts)

Solution 1:

If we build a bridge between city i and X(i), then for all cities j<i, we try to most as many bridges as possible such that X(j)<X(i) to avoid crossing.

Define OPT(i) as maximum number of bridges built for southern cities from 1 to i, while guaranteeing that there is a bridge is built between city i and X(i).

Base case: OPT(1) = 1

Recursive relation (for i>1):
OPT(i) = max_$_{\{k:\ 1<=k<i,\ X(k)<X(i)\}}$ {OPT(k)} +1

Max_bridge_num = max_$_{\{i\}}$ {OPT(i)}

Solution 2:
Define OPT(n,m) as the maximum number of no crossing bridges that can be built between southern cities 1,,,,n and northern cities 1,…, m.

Base case: $OPT(1,m) = 0$, if $m<X(1)$, $OPT(1,m)=1$, if $m>=X(1)$;
$OPT(n,1) = 0$, if $n<X^{-1}(1)$, $OPT(n,1) = 0$, if $n>=X^{-1}(1)$

Recursive relation:

$OPT(n,m) = OPT(n-1, m-1)+1$, if $m = X(n)$
$OPT(n,m) = \max\{OPT(n-1,m), OPT(n,m-1)\}$, if $m \mathrel{!}= X(n)$

a) Provide the algorithm to find the value of the optimal solution using the above recurrence formula (6 pts)

Solution 1:

```
For i = 1  to N
    Compute OPT(i) according to a)
End

Find the maximum among OPT(1), OPT(2), …..OPT(N)
```

Solution 2:

```
OPT(1) = 1;
For i = 2 to N
    L(i) = 0;
    For k = 1:i-1
        If( (X(k)<X(i)))
            L(i) = k;
        End
    End
    OPT(i) = max{ OPT(L(i)) + 1, OPT(i-1) }
End
```

Solution 2:

```
Compute Base case;
For n = 2:N
    For m = 2: N
        Compute OPT according to recursive relation.
    End
End
Returen OPT(N,N)
```

b) Provide the algorithm to list the actual bridges corresponding to the value found in part b (7 pts)

Solution 1:

Method 1:

```
Denote i* = argmax_i {OPT(i)}

Bridge_list = i*;

If(i*==1)
          Return Bridge_list;
End

For j = i* to 1

          For k = 1:j-1
                    If( (X(k)<X(j)) &&(OPT(j)==OPT(k)+1) )
                              Bridge_list = [k, Bridge_list];
                    End
          end
 End
 Return Bridge_list;
```

Method 2:

If you keep recording the corresponding k which achieves the maximum of OPT(i) in part b), denote it as p(i); if no k achieves the maximum, i.e., OPT(k)=0, then p(i) = i.

```
Denote i* = argmax_i {OPT(i)}
Bridge_list = i*;

If (i*==1)
          Return Bridge_list
End

j= i*;
While(p(j)!=j)
          Bridge_list = [p(j), Briedge_list];
End
Return Bridge_list.
```

Complexity: O(n^2)

Solution 2:

Compute Build_bridge (N,N)

Bridge = [];

```
Build_bridge (n,m)
{
    If n ==1
        If m>=X(1)
            Return [Bridge, (1,X(1))];
        Else
            Return Bridge;
    End

    If m = 1
        If m == X(n)
            Return [Bridge, (n,1)]
        Else
            Return Bridge
    End

    If OPT(n,m) == OPT(n-1,m-1) + 1
        Bridge = [Bridge,(n,m)];
        Buildbridge(n-1,m-1)
    Elseif OPT(n,m) == OPT(n-1,m)
        Buildbridge(n-1,m);
    Else
        Brildbridge(n,m-1);
    End

}
```

4) 20 pts
There are n reservoirs and m cities. Due to the drought you have to bring water from the reservoirs to the cities through a network of pipes. Each city *i* has a request for *Ci* gallons of water every day. To make matters worse, engineers have detected a leaking pipe in the water network. If the leaking pipe is used, they will lose L gallons of water per day through that pipe regardless of how much water flows through it (i.e., as long as the pipe is used, it leaks L gallon, we cannot control the leak by trying to push less water through that pipe). The other option is to shut down the leaking pipe at both ends, but that might reduce the capacity of their network. The water network is represented by a graph. Each edge represents the capacity of the pipe in gallons per day. Provide algorithms to determine if it is possible to:

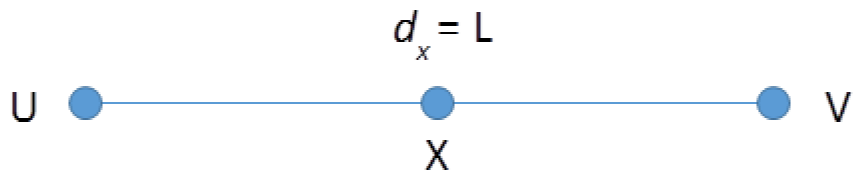
a) Meet all demands for water at all cities without using the leaking pipe (7 pts)

Assume the original graph is G=(V,E). Build a new graph with a supersource s and a supersink t.
Connect the supersource to each reservoir with an edge of infinity capacity. Connect city i to supersink with capacity C_i
Remove the edge corresponding to the leaking pipe. Run a max flow from s to t. If the flow over the edge from city i to supersink t equals to C_i for all city i. Then the demand of all city can be met.

b) Meet all demands for water at all cities by using the leaking pipe (6 pts)

The easiest way to model the second subproblem would be to:

- split the edge representing that segment of the pipe with a leak into two edges
- The point connecting these two edges will represent the location of the leak
- Each of the two segments will have the same capacity as the original leaking pipe
- We place a demand of L units at the location of the leak
- The rest is similar to how we dealt with demand for flow in part, i.e. connet all nodes with demand including X to supersink, etc. and solve for max flow, etc.



c) Meet all demands for water at all cities after fixing the leaking pipe (6 pts)

Pretty similar with solution of a), this time we don't have to remove the leaking pipe. Just use it as a usual edge. Then use the same algorithm in solution a)

5) 20 pts
Assume you want to ski down the mountain. You want the total length of your run to be as long as possible, but you can only go down, i.e. you can only ski from a higher position to a lower position. The height of the mountain is represented by an n X n matrix A. A[i][j] is the height of the mountain at position (i,j). At position (i,j), you can potentially ski to four adjacent positions (i-1,j) (i,j-1), (i,j+1), and (i+1,j) (only if the adjacent position is lower than current position). Movements in any of the four directions will add 1 unit to the length of your run. Provide a dynamic programming solution to find the longest possible downhill ski path starting at any location within the given n by n grid.

Define OPT(i,j) be the longest path start from position (i,j).

The recurrence is

If there are positions (i’,j') adjacent position of (i,j) with height less than A[i][j]
OPT(i,j) = max_{(i’,j’) is adjacent to (i,j), A[i’][j’] < A[i][j]} OPT(i’,j’) + 1,
If not, OPT(i,j) = 0 (boundary condition).

==
We solve it by memorization:
pseudocode:

```
search(i,j):
if OPT(i,j) >= 0
  return OPT(i,j)
else
  max = 0
  if A[i-1][j] < A[i][j] and search(i-1, j) +1> max
    max =search(i-1, j) +1
  if A[i+1][j] < A[i][j] and search(i+1, j) +1> max
    max =search(i+1, j) +1
  if A[i][j-1] < A[i][j] and search(i, j-1) +1> max
    max =search(i, j-1) +1
  if A[i][j+1] < A[i][j] and search(i, j+1) +1> max
    max =search(i, j+1) +1
  OPT(i,j) = max
  return OPT(i,j)

find_path(i,j):
if OPT(i,j) == 0:
  return [(i,j)]
else
  if OPT(i,j) = OPT(i-1,j) + 1
     return [(i,j), find_path(i-1,j)]
```

```
   if OPT(i,j) = OPT(i+1,j) + 1
       return [(i,j), find_path(i+1,j)]
   if OPT(i,j) = OPT(i,j-1) + 1
       return [(i,j), find_path(i,j-1)]
   if OPT(i,j) = OPT(i,j+1) + 1
       return [(i,j), find_path(i,j+1)]

OPT(i,j) = -1 for all  1<= i , j <=n
result = 0
for i = 1 to n
 for j = 1 to n
  if search(i,j) > max
    max = search(i,j)
    start = (i,j)

return find_path(start)
```
==

The time complexity is O(n^2) because we have n^2 subproblems and solve each of them will require O(1) time.

==

You can also solve the problem iteratively as follows:

```
update(i,j):

   max = 0

   if A[i-1][j] < A[i][j] and OPT(i-1, j) +1> max

      max =OPT(i-1, j) +1

   if A[i+1][j] < A[i][j] and OPT(i+1, j) +1> max

      max =OPT(i+1, j) +1

   if A[i][j-1] < A[i][j] and OPT(i, j-1) +1> max

      max =OPT(i, j-1) +1

   if A[i][j+1] < A[i][j] and OPT(i, j+1) +1> max

      max =OPT(i, j+1) +1
```

```
  return max

for (i,j) in increasing order of A[i][j]
  OPT(i,j) = update(i,j)
```

The path finding procedure is the same as the recursive one.

The time complexity is O(n^2log n) because of sorting.