



## CSE 4131: ALGORITHM DESIGN 2

### Assignment 1: Solution

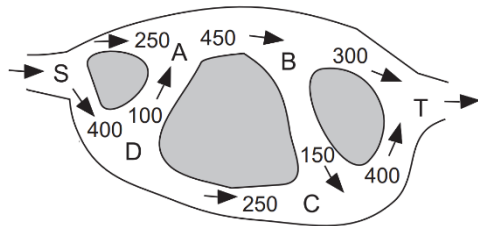
CO 1: to understand the Network flow problem and apply it to solve different real-world problems.

Sl. No.	Questions	PO	level
1.	<p>Consider the maximum flow in a given network between two designated nodes <math>s</math> and <math>t</math>. For each of the following statements, either explain why it is true or provide a counterexample.</p> <p>(a) If the capacity of every edge is even, then the value of the maximum flow must be even.</p> <p>(b) If the capacity of every edge is even, then there is a maximum flow in which the flow on each edge is even.</p> <p>(c) If the capacity of every edge is odd, then the value of the maximum flow must be odd.</p> <p>(d) If the capacity of every edge is odd, then there is a maximum flow in which the flow on each edge is odd.</p> <p><b>Solution:</b></p> <p>(a) True: If the capacity of every edge is even, then the value of the maximum flow must be even. This is because in a flow network, the maximum flow value is determined by the sum of flows through individual edges. Since the capacities are all even, any combination of flows through the edges that reaches the maximum flow value must also be even. This is because the flow along each edge contributes an even amount to the total flow value.</p> <p>(b) True: let's consider a flow network where the capacities of all edges are even integers. When you send flow along a path from the source to the sink, the amount of flow sent along each edge must be an integer and, importantly, since the capacity of each edge is even, the flow along each edge will also be even. if we think about the conservation of flow at each node (except the source and sink), the sum of the flow entering the node must be equal to the sum of the flow leaving the node. Since the flow along each edge is even, the total flow entering or leaving a node will be even.</p> <p>(c) False: If the capacity of every edge is odd, the value of the maximum flow may or may not be odd. Counterexample: Consider a flow network where all capacities are odd, but the maximum flow value is even. For instance, a network with capacities 1, 1, 1, and 1 on a path from <math>s</math> to <math>t</math>. Here, the maximum flow is 2.</p> <p>(d) False: No, the statement is not necessarily true. If the capacities of every edge are odd, it does not imply that there is a maximum flow in which the flow on each edge is odd. Consider the following counterexample: Let's say we have a simple flow network with two nodes, a source (<math>S</math>) and a sink (<math>T</math>), connected by a single edge with an odd capacity of 1. <math>(S) \rightarrow (T)</math>, capacity=1. In this case, the maximum flow that can be sent from <math>S</math> to <math>T</math> is 1. However, since the capacity of the edge is odd, the maximum flow along this edge is also odd. There is no way to send an odd flow along this edge without violating the capacity constraint. This counterexample demonstrates that while the capacities of every edge are odd, it doesn't necessarily guarantee that there exists a maximum flow in which the flow on each edge is odd.</p>		



2.	<p>Suppose you are given a directed graph <math>G = (V, E)</math>, with a positive integer capacity <math>C_e</math> on each edge <math>e</math>, a designated source <math>s \in V</math>, and a designated sink <math>t \in V</math>. You are also given an integer maximum <math>s</math>-<math>t</math> flow in <math>G</math>, defined by a flow value <math>f_e</math> on each edge <math>e</math>.</p> <p>Now suppose we pick a specific edge <math>e \in E</math> and increase its capacity by one unit. Show how to find a maximum flow in the resulting capacitated graph in time <math>O(m + n)</math>, where <math>m</math> is the number of edges in <math>G</math> and <math>n</math> is the number of nodes.</p> <p><b>Solution:</b></p> <p>The point here is that <math>O(m + n)</math> is not enough time to compute a new maximum flow from scratch, so we need to figure out how to use the flow <math>f</math> that we are given. Intuitively, even after we add 1 to the capacity of edge <math>e</math>, the flow <math>f</math> can't be that far from maximum; after all, we haven't changed the network very much. In fact, it's not hard to show that the maximum flow value can go up by at most 1</p> <p><b>Claim:</b> Consider the flow network <math>G</math> obtained by adding 1 to the capacity of <math>e</math>. The value of the maximum flow in <math>G</math> is either <math>v(f)</math> or <math>v(f) + 1</math>.</p> <p><b>Proof:</b></p> <p>The value of the maximum flow in <math>G</math> is at least <math>v(f)</math>, since <math>f</math> is still a feasible flow in this network. It is also integer-valued. So it is enough to show that the maximum-flow value in <math>G'</math> is at most <math>v(f) + 1</math>.</p> <p>By the Max-Flow Min-Cut Theorem, there is some <math>s</math>-<math>t</math> cut <math>(A, B)</math> in the original flow network <math>G</math> of capacity <math>v(f)</math>. Now we ask: What is the capacity of <math>(A, B)</math> in the new flow network <math>G'</math>? All the edges crossing <math>(A, B)</math> have the same capacity in <math>G'</math> that they did in <math>G</math>, with the possible exception of <math>e</math> (in case <math>e</math> crosses <math>(A, B)</math>). But <math>C_e</math> only increased by 1, and so the capacity of <math>(A, B)</math> in the new flow network <math>G'</math> is at most <math>v(f) + 1</math>.</p> <p>The above statement suggests a natural algorithm. Starting with the feasible flow <math>f</math> in <math>G'</math>, we try to find a single augmenting path from <math>s</math> to <math>t</math> in the residual graph <math>G_f'</math>. This takes time <math>O(m + n)</math>. Now one of two things will happen. Either we will fail to find an augmenting path, and in this case we know that <math>f</math> is a maximum flow. Otherwise the augmentation succeeds, producing a flow <math>f</math> of value at least <math>v(f) + 1</math>. In this case, we know that <math>f</math> must be a maximum flow. So either way, we produce a maximum flow after a single augmenting path computation.</p>		

3. This diagram represents a road network. All vehicles enter at S and leave at T. The numbers represent the maximum flow rate in vehicles per hour in the direction from S to T.
- What is the maximum number of vehicles which can enter and leave the network every hour?
  - Which single section of road could be improved to increase the traffic flow in the network?



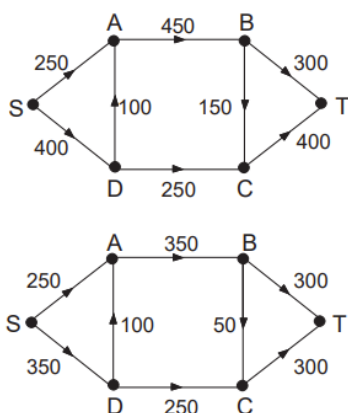
**Solution:**

The network in the previous activity can be more easily analysed when drawn as a graph, as shown

The arrows show the flow direction; consequently this is called a directed graph or di-graph. In this case the edges of the graph also have capacities : the maximum flow rate of vehicles per hour. The vertices S and T are called the source and sink, respectively

You should have found that the maximum rate of flow for the network is 600. This is achieved by using each edge with flows as shown.

Also Section DA should be improved by a capacity of 150 in total to increase the total network flow.

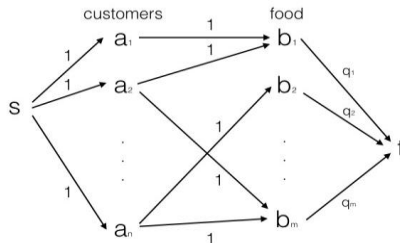


4. The Jungle Junket food truck produces a large variety of different lunch menu items. Unfortunately, they can only produce their foods in limited quantities, so they often run out of popular items, making customers sad. To minimize sadness, Jungle Junket is implementing a sophisticated lunch-ordering system. Customers text in their acceptable choices before lunch time. Then they can use an algorithm to preassign lunches to customers. Customers who do not get one of their choices should receive a Rs100 voucher. Jungle Junket would like to minimize the number of vouchers they give out.

Give an efficient algorithm for Jungle Junket to assign lunches to customers. In general, suppose that, on a given day, Jungle Junket has produced  $m$  types of food items  $b_1, \dots, b_m$ , and the quantity of each type of food item  $b_j$  is exactly  $q_j$ . Suppose that  $n$  customers  $a_1, \dots, a_n$  text in their preferences, where each customer  $a_i$  submits a set  $A_i$  of one or more acceptable lunch choices. The algorithm should assign each customer either one of his/her choices or a Rs100 voucher. It should minimize the number of vouchers.  
(Hint: Model this as a max flow problem.)

**Solution:**

Model this as a max flow problem. Define a flow network as follows. in which  $a_1$  prefers  $b_1$ ,  $a_2$  prefers  $b_1$  and  $b_m$ ,...  $a_n$  prefers  $b_2$  and  $b_m$ .



Include special vertices  $s$  and  $t$  as usual. Have a “first layer” of  $n$  vertices corresponding to the customers, and a “second layer” of  $m$  vertices corresponding to the foods. Include an edge from  $s$  to each customer, with capacity 1. Include an edge from customer  $a_i$  to food item  $b_j$  exactly if  $j \in A_i$ ; this will also have capacity 1. Include an edge from food item  $b_j$  to  $t$  with capacity  $q_j$ .

**Proof:** A flow  $f$  on this network yields an assignment of food items to customers that satisfies the customer and food quantity constraints. Specifically, for each customer  $a_i$ , if some edge  $f(a_i, b_j)$  has flow 1, then assign food item  $b_j$  to customer  $a_i$ . Note that each customer  $a_i$  can get at most one food item, because  $a_i$  has incoming flow at most 1, so its outgoing flows must also total at most 1. Since all flows are integral, only one outgoing edge can have positive flow. Also note that each food item  $b_j$  cannot be assigned to more than  $q_j$  customers, because  $b_j$  has outgoing flow at most  $q_j$ , so its incoming flows must also total at most  $q_j$ . Thus, the food assignment arising from flow  $f$  satisfies all the customer and food constraints.

Conversely, any food assignment satisfying all these constraints corresponds directly to a flow through the network: Assign flow 1 to edge  $(a_i, b_j)$  exactly if customer  $a_i$  gets assigned food item  $b_j$ . Assign flows to the edges from  $s$  and to  $t$  to achieve flow conservation.

Moreover, a max flow through this network satisfies the maximum number of customers, because the definition of a max flow says that it maximizes the total flow out of  $s$  (which corresponds to the number of customers satisfied).

**Algorithm:** So, all we need to do is run a max flow algorithm on this network, produce an integral max flow  $f$ , and interpret it as a food assignment. The maximum number of satisfied customers yields the minimum number of vouchers.

**Analysis:** The number of edges is  $E = O(mn)$ . The max flow  $|f| \leq n$ . Using the Ford-Fulkerson algorithm, the time complexity is  $O(E|f|) = O(mn^2)$ .

5. In this problem, you will design an algorithm that takes the following inputs:

- A flow network  $F = (G, c)$ , where  $G = (V, E)$  is a graph with source vertex  $s$  and target vertex  $t$ , and  $c$  is a capacity function mapping each directed edge of  $G$  to a nonnegative integer;
- A maximum flow  $f$  for  $F$ ; and
- A triple  $(u, v, r)$ , where  $u$  and  $v$  are vertices of  $G$  and  $r$  is a nonnegative integer  $= c(u, v)$ .

The algorithm should produce a maximum flow for flow network  $F' = (G, c')$ , where  $c'$  is



identical to  $c$  except that  $c(u, v) = r$ . The algorithm should run in time  $O(k \cdot (V + E))$ , where  $|c(u, v) - r| = k$ . The algorithm should behave differently depending on whether  $r > c(u, v)$  or  $r < c(u, v)$ .

(a) Start by proving the following basic, general results about flow networks:

1. Increasing the capacity of a single edge  $(u, v)$  by 1 can result in an increase of at most 1 in the max flow.
2. Increasing the capacity of a single edge  $(u, v)$  by a positive integer  $k$  can result in an increase of at most  $k$  in the max flow.
3. Decreasing the capacity of a single edge  $(u, v)$  by 1 can result in a decrease of at most 1 in the max flow.
4. Decreasing the capacity of a single edge  $(u, v)$  by a positive integer  $k$  can result in a decrease of at most  $k$  in the max flow.

Solution:

1. If  $(u, v)$  is in every min cut, then increasing the capacity of  $(u, v)$  by 1 increases the min cut value by 1. If  $(u, v)$  is not in every min cut, then increasing the capacity of  $(u, v)$  by 1 leaves the min cut value unchanged. Either way, the capacity increases by at most 1. The claim follows from the max-flow-min-cut theorem.

2. Increasing by  $k$  is the same as increasing in steps of 1. By part 1, each such step increases the max flow by at most 1. So the total increase is at most  $k$ .

3. If  $(u, v)$  is in some min cut, then decreasing the capacity of  $(u, v)$  decreases the min cut value by 1. If  $(u, v)$  is not in any min cut, then decreasing the capacity of  $(u, v)$  by 1 leaves the min cut value unchanged. Either way, the capacity decreases by at most 1. The claim follows from the max-flow-min-cut theorem.

4. Decreasing by  $k$  is the same as decreasing in steps of 1. By part 4, each such step decreases the max flow by at most 1. So the total decrease is at most  $k$ .

(b) Suppose that  $r > c(u, v)$ . Describe your algorithm for this case in detail, prove that it works correctly, and analyze its time complexity (in terms of  $V$ ,  $E$ , and  $k$ ).

Solution:

Then the max flow might increase; by Part (a) 2, it increases by at most  $k$ .

Algorithm: Regard the existing flow  $f$  as a flow in the new flow network  $F' = (G, c')$ .

Start with the residual network of  $F'$  for flow  $f$ . Repeat  $k$  times:

1. Look for an augmenting path in the residual network.
2. If you find one, add it to the existing flow, else return.

Analysis: Using DFS to search for an augmenting path, each pass through the loop takes time  $O(V + E)$ , so the total time is  $O(k \cdot (V + E))$ .

Proof: As noted, increasing the capacity of a single edge by  $k$  can result in an increase of at most  $k$  in the value of the max flow. Each time we find an augmenting path, it increases the flow by at least 1. So within at most  $k$  tries, we reach the max flow.

(c) Suppose that  $r < c(u, v)$ . Describe your algorithm for this case in detail, prove that it works correctly, and analyze its time complexity.

Solution:

Then the max flow might decrease; by Part (a) 4, it decreases by at most  $k$ .

Algorithm: The algorithm consists of two phases, first removing some flow to fit the reduced capacity of the new flow network  $F' = (G, c')$ , and then restoring flow to achieve the new max flow.

In Phase 1, if  $f(u, v) \leq r$  then we do not reduce any flows. Otherwise, we reduce the flow on  $(u, v)$  one unit at a time, until it reaches  $r$ . For each unit, we proceed as follows.

First reduce the flow on  $(u, v)$  by 1. Then, using DFS, search backwards from vertex  $u$ , following incoming edges with positive (incoming) weight, looking for a simple reverse



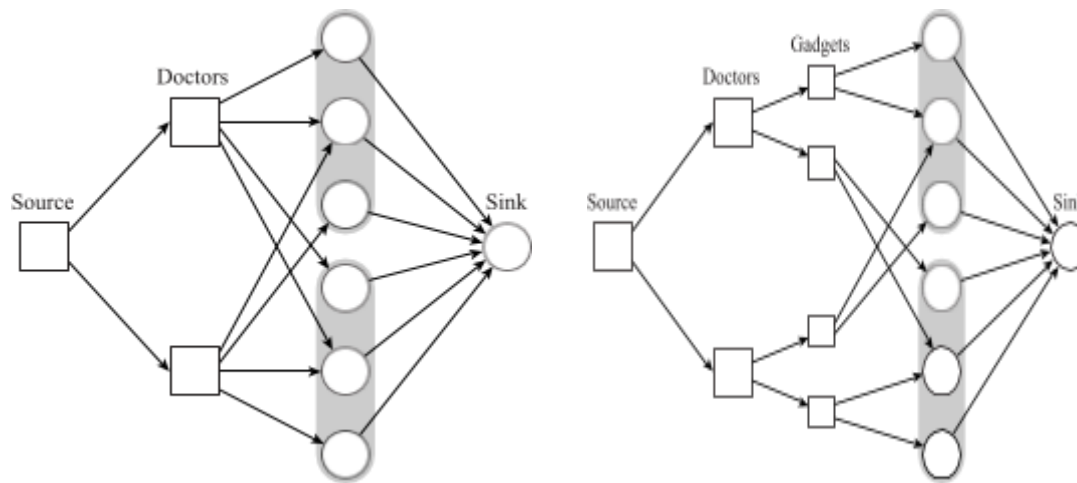
	<p>path to <math>s</math>, <math>t</math>, or <math>v</math>. To see why such a path exists, proceed step by step from <math>u</math>, each time following some incoming edge with positive (incoming) weight. Such an edge must exist unless we have reached one of the three listed vertices, because we started with a flow satisfying the flow conservation property. Subtract 1 from the flows along each of the edges in this path.</p> <p>If the path we found ended in <math>v</math>, we have already restored the flow conservation condition throughout the network. Otherwise, vertex <math>v</math> still does not satisfy flow conservation. So in this case, we use DFS again to search forwards from <math>v</math>, following outgoing edges with positive (outgoing) weight, looking for a simple path to <math>t</math> or <math>s</math>. Subtract 1 from the flows along each of the edges in this path.</p> <p>At this point, we have restored the flow conservation condition while reducing the flow on <math>(u, v)</math> by 1. We repeat this a total of <math>f(u, v) - r</math> times, reducing the flow on <math>(u, v)</math> to <math>r</math> and restoring the flow conservation condition.</p> <p>The result of this reduction is now a valid flow for the new network <math>F' = (G, c')</math>; however, it needs not to be maximum.</p> <p>In Phase 2, we augment it to restore maximality. We proceed as in Part (b), trying <math>k</math> times to augment the flow from <math>s</math> to <math>t</math>.</p> <p>Analysis: In Phase 1, each reduction of capacity by 1 takes time <math>O(V + E)</math>, using DFS for the two searches. Thus, Phase 1 takes time <math>O(k \cdot (V + E))</math>. Phase 2 is like Part (b), and so takes time <math>O(k \cdot (V + E))</math>.</p> <p>Proof: As we noted, after Phase 1, we have a valid flow. Moreover, in each reduction, we have reduced the value of the flow by at most 1. Therefore, in all, we have reduced the value of the flow by at most <math>k</math>.</p> <p>The value of the max flow can be anywhere between the value of the flow after Phase 1 and the value of the original max flow. The difference between these two values is at most <math>k</math>, so <math>k</math> iterations are enough to restore the max.</p>		
7.	<p>You are helping the medical consulting firm Doctors Without Weekends set up the work schedules of doctors in a large hospital. They've got the regular daily schedules mainly worked out. Now, however, they need to deal with all the special cases and, in particular, make sure that they have at least one doctor covering each vacation day.</p> <p>Here's how this works. There are <math>k</math> vacation periods (e.g., the week of Christmas, the July 4th weekend, the Thanksgiving weekend, . . . ), each spanning several contiguous days. Let <math>D_j</math> be the set of days included in the <math>j</math>th vacation period; we will refer to the union of all these days, <math>\cup_j D_j</math>, as the set of all vacation days.</p> <p>There are <math>n</math> doctors at the hospital, and doctor <math>i</math> has a set of vacation days <math>S_i</math> when he or she is available to work. (This may include certain days from a given vacation period but not others; so, for example, a doctor may be able to work the Friday, Saturday, or Sunday of Thanksgiving weekend, but not the Thursday.)</p> <p>Give a polynomial-time algorithm that takes this information and determines whether it is possible to select a single doctor to work on each vacation day, subject to the following constraints.</p> <ul style="list-style-type: none"><li>• For a given parameter <math>c</math>, each doctor should be assigned to work at most <math>c</math> vacation days total, and only days when he or she is available.</li><li>• For each vacation period <math>j</math>, each doctor should be assigned to work at most one of the days in the set <math>D_j</math>. (In other words, although a particular doctor may work on several vacation days over the course of a year, he or she should not be assigned to work two or more days of the Thanksgiving weekend, or two or more days of the July 4th weekend, etc.)</li></ul> <p>The algorithm should either return an assignment of doctors satisfying these constraints or report (correctly) that no such assignment exists.</p> <p>Solution:</p> <p>This is a very natural setting in which to apply network flow, since at a high level we're trying to match one set (the doctors) with another set (the vacation days). The complication comes from the requirement that each doctor can work at most one day in each vacation</p>		



period.

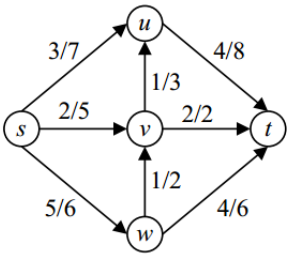
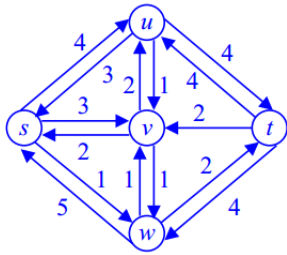
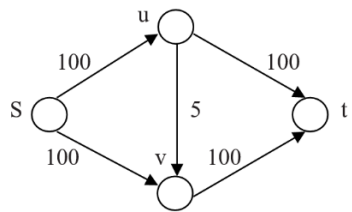
So to begin, let's see how we'd solve the problem without that requirement, in the simpler case where each doctor  $i$  has a set  $S_i$  of days when he or she can work, and each doctor should be scheduled for at most  $c$  days total. The construction is pictured in Figure (a). We have a node  $u_i$  representing each doctor attached to a node  $v_d$  representing each day when he or she can work; this edge has a capacity of 1. We attach a super-source  $s$  to each doctor node  $u_i$  by an edge of capacity  $c$ , and we attach each day node  $v_d$  to a super-sink  $t$  by an edge with upper and lower bounds of 1. This way, assigned days can "flow" through doctors to days when they can work, and the lower bounds on the edges from the days to the sink guarantee that each day is covered. Finally, suppose there are  $d$  vacation days total; we put a demand of  $+d$  on the sink and  $-d$  on the source, and we look for a feasible circulation. (Recall that once we've introduced lower bounds on some edges, the algorithms in the text are phrased in terms of circulations with demands, not maximum flow.)

But now we have to handle the extra requirement, that each doctor can work at most one day from each vacation period. To do this, we take each pair  $(i, j)$  consisting of a doctor  $i$  and a vacation period  $j$ , and we add a "vacation gadget" as follows. We include a new node  $w_{ij}$  with an incoming edge of capacity 1 from the doctor node  $u_i$ , and with outgoing edges of capacity 1 to each day in vacation period  $j$  when doctor  $i$  is available to work. This gadget serves to "choke off" the flow from  $u_i$  into the days associated with vacation period  $j$ , so that at most one unit of flow can go to them collectively. The construction is pictured in Figure (b). As before, we put a demand of  $+d$  on the sink and  $-d$  on the source, and we look for a feasible circulation. The total running time is the time to construct the graph, which is  $O(nd)$ , plus the time to check for a single feasible circulation in this graph.



(a) Doctors are assigned to holiday days without restricting how many days in one holiday a doctor can work. (b) The flow network is expanded with "gadgets" that prevent a doctor from working more than one day from each vacation period. The shaded sets correspond to the different vacation period

8. A graph has a unique minimum cut if there is only one cut that whose weight is the minimum. Design an algorithm that finds if a graph has a unique minimum cut.  
Solution:  
First compute a minimum s-t cut  $C$ , and define its volume by  $|C|$ . Let  $e_1, e_2, \dots, e_k$  be the edges in  $C$ . For each  $e_i$ , try increasing the capacity of  $e_i$  by 1 and compute a minimum cut in the new graph. Let the new minimum cut be  $C_i$ , and denote its volume (in the new graph) as  $|C_i|$ . If  $|C| = |C_i|$  for some  $i$ , then clearly  $C_i$  is also a minimum cut in the original graph and  $C \neq C_i$ , so the minimum cut is not unique. Conversely, if there is a different minimum cut  $C'$  in the original graph, there will be some  $e_i \in C$  that is not in  $C'$ , so increasing the capacity of that edge will not change the volume of  $C'$ , thus  $|C| = |C_i|$ . In

	<p>conclusion, the graph has a unique minimum cut if and only if <math> C  &lt;  C_i </math> for all <math>i</math>. The algorithm takes at most <math>m + 1</math> computing of minimum cuts, and therefore runs in polynomial time.</p>		
9.	<p>a. Draw the residual graph for the given graph, in each edge <math>f(e)/c(e)</math>, <math>f(e)</math> represents the flow and <math>c(e)</math> represents the capacity in that edge.</p>  <p>Solution:</p>  <p>b. Given a graph G, determine the maximum and minimum number of augmentations possible for the Ford-Fulkerson algorithm to compute the maximum flow from a source vertex s to a sink vertex t (path may repeat)</p>  <p>Solution:</p> <p>The minimum number of augmentations to get the max flow using Ford Fulkerson is 2 (sut, svt).</p> <p>The maximum number of augmentations to get the max flow using Ford Fulkerson is <math>200/5=40</math>. (suv and svut alternatively).</p>		
10.	<p>ITER, SOA requires an algorithm to schedule mid-semester exams for their courses each semester. There are <math>n</math> courses offered, <math>r</math> available rooms, and <math>m</math> time slots for exams. Given arrays <math>E[1...n]</math> and <math>S[1...r]</math>, <math>E[i]</math> represents the number of students enrolled in <math>i</math>-th course and <math>S[j]</math> represents the number of seats in each room respectively, an exam for course <math>i</math> can only be scheduled in room <math>j</math> if the number of enrolled students for that course is less than or equal to the number of seats available in the room (<math>E[i] \leq S[j]</math>). It's assumed that no two courses have overlapping enrollments. The goal is to develop an efficient algorithm for ITER to assign a room and a time slot to each course, or report if no such assignment is possible.</p> <p>Solution: Create three sets of vertices, one set <math>C</math> for each course, one set <math>R</math> for each room, and one set <math>T</math> for each slot.</p> <p>Also create a source <math>s</math> and a sink <math>t</math>.</p> <p>Add an edge from <math>s</math> to each vertex in <math>C</math> with capacity 1.</p> <p>Add an edge from vertex <math>i</math> in <math>C</math> to vertex <math>j</math> in <math>R</math> with capacity 1 if and only if <math>E[i] \leq S[j]</math>.</p>		





	Add an edge from each vertex in R to each vertex in T with capacity 1. Add an edge from each vertex in T to sink t with capacity $\geq r$ . Now find maximum flow. It is possible only if all edges from s are saturated (or equivalently, the maximum flow in n).		



*Dept. of Comp. Sc. and Engg., ITER, S'O'A Deemed To Be University*