Solutions to the Practice Midterm Problems

(Updated: Tue, Apr 1.)

Solution 1:

(a) $\Theta(n)$: The innermost loop is executed i times, and each time the value of i is halved. So the overall running time is

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \;=\; n\left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) \;\leq\; 2n \;=\; \Theta(n).$$

(b) $kn/2$: The sum of degrees of the vertices of such a degree-k graph is at most kn. The sum of degrees in a graph is twice the number of edges (since each edge is counted twice, one for each endpoint). Therefore, the number of edges is at most $kn/2$.

(c) $d[v] < d[u] < f[u] < f[v]$: This follows from the DFS Parentheses Lemma. Here is a direct argument. By the recursive nature of DFS, descendants (and, in particular, children) are discovered later and finish earlier.

(d) False. After offsetting, the cost of each path is biased by the added value C multiplied by the number of edges on the path. Thus, two paths of equal total weight, but which have different numbers of edges would have affected differently. This implies that shortest paths are not preserved.

(e) It has at least one negative-cost cycle: Bellman-Ford converges under the assumption that the graph has no negative cost cycles. In particular, the d-values along any negative cost path (and any vertices reachable from this cycle) just get smaller and smaller as the algorithm iterates.

(f) All four statements hold: Because Gonzalez only adds (never removes) centers, as more centers are added, the Δ_i and Γ_i values can only get smaller (or stay the same). When a new center is added in some iteration, it is placed at the point that has the maximum distance to its closest center. This implies that $\Gamma_{i+1} \leq \Delta_i$, and hence $\Gamma_4 \leq \Delta_3$. In Lecture 6 (Claim 2), it was proved $\Gamma_i \geq \Delta_{i-1}$, and hence $\Gamma_4 \geq \Delta_3$.

(g) The max flow value will be evenly divisible by 3, and further, (if it is computed using path-augmentation, like Ford-Fulkerson) the flow values on each edge will also be evenly divisible by 3. To see this, just divide all the capacities by 3. They are still integers, and it is known that the Ford-Fulkerson algorithm generates an optimal, integer-valued flow. To return to the original problem, multiply all flow values by 3.

Solution 2: This is hard to do by a direct application of DFS. There are a couple ways to do this. First, we could apply topological sorting, and then process the vertices in increasing topological order. We initialize $L[v] \leftarrow 0$ for all vertices. Whenever we visit a vertex u, for each of its neighbors v, we set $L[v] \leftarrow \max(L[v], L[u] + 1)$. Both the topological sort and this subsequent

processing take $O(n+m)$ time. This works, because whenever we visit any vertex v, we know that all its predecessors have been visited, and hence their L-values are fixed.

Here is an alternative approach based on DFS. We begin by computing the edge-reversed graph G^R. We have indicated in class that this can be constructed in $O(n+m)$ time, by a simple traversal of the adjacency list. In G^R, $L[v]$ is the length of the longest path that starts at v. We can compute this by DFS. In particular, when we visit each vertex u, we apply DFS recursively to all its neighbors v. On return, their L-values are fixed, and so we set $L[u] \leftarrow \max(L[u], L[v]+1)$. The DFSvisit portion of the algorithm is given in the following code block. As a standard application of DFS, it runs in $O(n+m)$ time.

Longest path emanating from each vertex

```
longest-path(u) {                          // longest path starting at u
    mark[u] = discovered
    L[u] = 0
    for each (v in Adj(u)) {               // visit all of u's neighbors
        if (mark[v] == undiscovered)
            longest-path(v)                // visit it if necessary
        L[u] = max(L[u], 1 + L[v])         // update u's longest path
    }
}
```

Solution 3: Define $P[u]$ to be the number of maximal paths that start at u. We apply DFS to G and compute $P[u]$ as we visit each vertex u. If u has no outgoing edges then $P[u] = 1$, which counts the trivial path $\langle u \rangle$. Otherwise, observe that we can form every maximal path from u by taking a maximal path from each of its neighbors, v, and prepending u to this path. Therefore, the total number of maximal paths is the sum of $P[v]$ for all neighbors v of u. The DFSvisit portion of the algorithm is presented in the following code block. Clearly, the running time is $O(n+m)$.

Number of maximal paths out of u

```
max-path-count(u) {                        // count maximal paths from u
     mark[u] = discovered
     if (Adj[u] == empty) P[u] = 1         // basis case - outdegree = 0
     else {                                // u is a non-terminal
        P[u] = 0
        for each (v in Adj[u]) {
            if (mark[v] == undiscovered)
                max-path-count(v)          // count paths from v
            P[u] += P[v]                   // add to u's path count
        }
     }
 }
```

Solution 4: We present a simple greedy algorithm. We go as far as possible before stopping to refuel. The variable `lastStop` indicates the location where we last got fuel. We find the farthest station from this that is within 100 miles and get fuel there. The code block below provides a sketch of the algorithm. To avoid subscripting out of bounds, let us assume that $x[n+1] = x[n]$.

```
fuel(x[1..n]) {
    lastStop = 0
    for (i = 1 to n) {
        if (x[i+1] > lastStop + 100) {  // will run out of gas before next station?
            add i to refuel list
            lastStop = x[i]             // this was our last gas
        }
    }
}
```

Clearly the running time is $O(n)$. Observe that this produces a feasible sequence, since we never go more than 100 miles before stopping. To establish optimality, let $F = \langle f_1, f_2, \ldots, f_k \rangle$ be the indices of an optimal sequence of refueling stops, and let $G = \langle g_1, g_2, \ldots, g_{k'} \rangle$ be the greedy sequence. If the two sequences are the same, then we are done. If not, let i be the smallest index such that $g_i \neq f_i$. Because greedy algorithm selects the *last* possible gas station, we know that $g_i > f_i$. Consider an alternative solution F' which comes by replacing f_i with g_i. We claim that F' is a also a feasible solution. To see this observe that sequence up to g_i is the same as G (which we know is feasible) and because we have delayed fueling, for the rest of the trip we have at least as much gas as we had with F (which we know is feasible). The sequence F' has the same number of stops as F and so is also optimal, and it has one more segment in common with G. By repeating this, eventually we will have an optimal solution that is identical to G.

Solution 5:

(a) The counterexample involves two files, one slightly longer but with much higher access probability. Let $(s_1, p_1) = (1, 0.1)$ and $(s_2, p_2) = (2, 0.9)$. If we put f_1 before f_2 (size order), the expected access cost is $1 \cdot 0.1 + (2 + 1) \cdot 0.9 = 2.8$, but if we reverse the order of files the cost is $2 \cdot 0.9 + (1 + 2) \cdot 0.1 = 2.1$, which is smaller (see Fig. 1(a)).

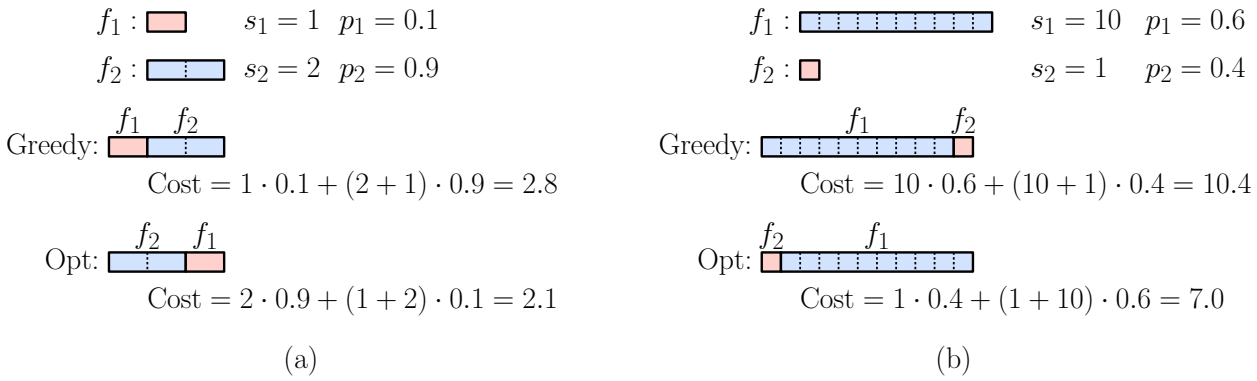

Figure 1: Solution to Problem 2(a) and (b).

(b) The counterexample involves two files, one slightly more likely to be accessed but with much larger size. Let $(s_1, p_1) = (10, 0.6)$ and $(s_2, p_2) = (1, 0.4)$. If we put f_1 before f_2 (decreasing probability order), the expected access cost is $10 \cdot 0.6 + (10 + 1) \cdot 0.4 = 10.4$, but if we reverse

the order of files the cost is $1 \cdot 0.4 + (1 + 10) \cdot 0.6 = 7.0$, which is smaller (see Fig. 1(b)).

(c) Intuitively, it seems smart to store the most frequently accessed files at the front of the tape, but it also makes sense to store the smallest files at the front of the tape. This suggests that the best way to store the files is in increasing order of s_i/p_i. Let us sort the files according to this statistic and lay them out in this order. (We will make the simplifying assumption that these ratios are distinct for all files.) To simplify notation, let us assume that the files have been renumbered, so that $s_1/p_1 < \cdots < s_n/p_n$. Clearly, this layout can be computed in $O(n \log n)$ time.

We will prove that this is optimal by contradiction. Suppose that the optimum layout O is different from the greedy layout. If so, there must be two consecutive files of the optimum layout that are not in sorted order. That is, we have $O = \langle \ldots, f_j, f_i, \ldots \rangle$, where $j > i$ according to our greedy order. Thus, we have $\frac{s_j}{p_j} > \frac{s_i}{p_i}$, or equivalently (because sizes and probabilities are both nonnegative), $p_j s_i - p_i s_j < 0$.

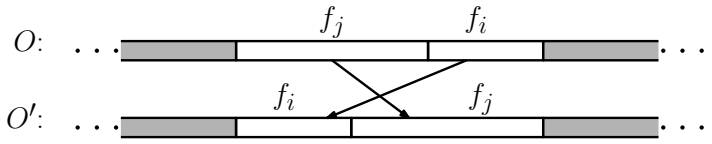

Figure 2: Solution to Problem 2(c), swapping f_j and f_i.

Let us consider how the cost changes if these two files are swapped in the layout (see Fig. 2). Call the resulting layout O'. After the swap, file f_j has moved s_i units towards the back of the tape, and so its individual access cost has increased by $p_j s_i$. Similarly, file i has moved s_j units closer to the front of the tape, so its individual access cost has decreased by $p_i s_j$. All the other files maintain their same placements on the tape, so there are no other changes affecting the total cost. Therefore, the net change in the total access cost is:

$$T(O') - T(O) \;=\; p_j s_i - p_i s_j \;<\; 0.$$

Therefore, $T(O') < T(O)$, which contradicts the optimality of O, and yields the desired contradiction.

Solution 6: Recall that for $0 \le i \le m$ and $0 \le j \le n$, we define the prefix strings $X_i = \langle x_1, \ldots, x_i \rangle$ and $Y_j = \langle y_1, \ldots, y_j \rangle$. For the same ranges of i and j, define $\text{lcsm}(i, j)$ to be the maximum weight achievable by matching X_i and Y_j.

For the basis case, if either i or j is zero, then clearly no matches are possible, and so $\text{lcsm}(i, j) = 0$. Otherwise (both i and j are at least 1), there are three cases.

(a) If $|x_i - y_j| \ge 2$, then no match is possible between these symbols, and hence either x_i is not used in the final match or y_j is not used in the match. As in LCS, we have $\text{lcsm}(i, j) = \max(\text{lcsm}(i-1, j), \text{lcsm}(i, j-1))$.

(b) If $|x_i - y_j| = 1$, then it might be that this is the best match or it might be that we should forgo this match in favor of finding a better match for either x_i or y_j (in case there may be an exact match possible). In the first case, we increase the weight by 1 and consume by x_i

and y_j. Otherwise, we eliminate either x_i or y_j. We consider all the options and take the best. Thus,

$$\text{lcsm}(i,j) \;=\; \max\Big(1+\text{lcsm}(i-1,j-1),\ \text{lcsm}(i-1,j),\ \text{lcsm}(i,j-1)\Big).$$

(c) If $x_i = y_j$, then we claim that there is no harm in assuming that these symbols are matched with each other. Suppose this were not so. If neither symbol is matched, then we could add this match and increase the total weight. On the other hand, if one of the symbols is matched, it must be matched earlier in the other sequence. By the subsequence structure, the other is not matched. By replacing that match with the $x_i \leftrightarrow y_j$ match, the total weight will either remain the same or increase.

By matching these symbols, the total weight increases by +2 and we consume both x_i and y_j, which implies $\text{lcsm}(i,j) = 2 + \text{lcsm}(i-1,j-1)$. (By the way, it would also be correct to include $\max(\text{lcsm}(i-1,j),\ \text{lcsm}(i,j-1))$ as part of this case, it is just not necessary.)

Combining these observations, we have the following DP formulation.

$$\text{lcsm}(i,j) \;=\; \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ \max(\text{lcsm}(i-1,j),\ \text{lcsm}(i,j-1)) & \text{if } i,j>0 \text{ and } |x_i - y_j| \geq 2, \\ \max(1+\text{lcsm}(i-1,j-1), & \\ \qquad \text{lcsm}(i-1,j), \text{lcsm}(i,j-1)) & \text{if } i,j>0 \text{ and } |x_i - y_j| = 1, \\ 2+\text{lcsm}(i-1,j-1) & \text{if } i,j>0 \text{ and } |x_i - y_j| = 0. \end{cases}$$

The final answer is $\text{lcsm}(m,n)$. As with the standard LCS problem, this can be implemented to run in $O(mn)$ time.

Solution 7:

(a) Let v_i be the cost of bottle i, and let b_i denote the number of pills it holds. In order to place W pills as inexpensively as possible, sort the bottles in increasing order of cost per pill, that is, v_i/b_i. Then fill the bottles in this order, until all the pills are gone.

To show that this is optimal, define the *incremental cost* for a pill to be v_i/b_i, where i denotes the bottle into which this pill was placed. Because we only pay for the portion of the bottle that we use, the total cost of bottling all the pills is equal to the sum of the incremental costs over all W pills. (This is important. For example, we can put a single pill into a last bottle, without paying for the entire bottle.)

For a given input, let G denote the sorted order of incremental costs for the greedy solution, and let O denote the sorted order of incremental costs for any optimal solution. We will use the usual exchange argument to show that G achieves the same cost as O.

If $G = O$, we are done. Otherwise, consider the first pill (in sorted order of incremental cost) where O differs from G. Let i denote the bottle into which G puts this pill, and let i' denote the bottle used by O. Since both sequences have been sorted, we know that O does not put any more pills into bottle i, even though there is still space remaining there (since G put this pill there). Since G places pills in increasing order of incremental cost, it must be that $v_i/b_i \leq v_{i'}/b_{i'}$. Let us create a new bottling plan by moving this one pill

from bottle i' to bottle i. The incremental change in cost by doing this is $v_i/b_i - v_{i'}/b_{i'} \leq 0$. Therefore, the total cost cannot increase as a result of the change. (Since O is optimal, it should not decrease.) By repeating this process, we will eventually convert O into G, while never increasing the bottling cost. Therefore, G is optimal.

(b) For $0 \leq i \leq n$, define $P(i, w)$ to be the minimum amount paid assuming that we place w pills using some subset of the first i bottles. For the basis case, observe that if $w = 0$ and $i = 0$, we can trivially put the 0 pills in 0 bottles for a cost of 0, and thus $P(0, 0) = 0$. If $W > 0$, then there is no solution using 0 bottles, and so we have $P(0, W) = \infty$.

For the induction, let us assume that $i \geq 1$. There are two cases. Either we do not use the ith bottle or we do. If not, we put all w pills in the first $i-1$ bottles, for a cost of $P(i-1, w)$. Otherwise, we put $\min(b_i, w)$ pills in this bottle, and put the remaining pills in the previous $i-1$ bottles. The total cost is $v_i + P(i-1, w - \min(b_i, w))$. We prefer the lower of the two choices, which implies the following recursive rule:

$$\begin{aligned} P(0, w) &= \begin{cases} 0 & \text{if } w = 0 \\ \infty & \text{otherwise} \end{cases} \\ P(i, w) &= \min(P(i-1, w), v_i + P(i-1, w - \min(b_i, w)) \qquad \text{for } i \geq 1. \end{aligned}$$

Solution 8:

(a) Suppose that the coin values are the form $\{1, 3, 9, \ldots, 3^i, \ldots\}$. The greedy algorithm simply uses the coin of the largest value until the remaining value is smaller than this coin value. For $0 \leq i \leq n$, let g_i denote the number of coins of value 3^i. The algorithm generates the maximum number of coins for the highest denomination and then decreases R by the amount generated.

- Let $i = \lfloor \log_3 R \rfloor$, and let $c = 3^i$. This is the smallest coin value that is less than or equal to R
- Repeat the following while $R > 0$:
  - While $R \geq c$, add c to the list of change and set $R \leftarrow R - c$.
  - Set $c \leftarrow c/3$.

Note that this will terminate, since eventually $c = 1$, and since R is an integer, it will eventually be reduced to zero. It is also easy to see that it produces valid change, since each time we add a coin of value c to the change, we reduce the remaining amount by c.

We will show that this greedy algorithm is optimal. We will prove this by induction on R. For the basis case, $R = 0$, there is no change to be generated, and any system (including greedy) is optimal. Otherwise, let $c = 3^i$ for $i \geq 0$ be the largest coin denomination that is less than or equal to R. Clearly, we cannot use any coins larger than c. We know that the greedy algorithm will use at least one coin of value c. We assert that the optimal algorithm must do the same. If so, both systems reduce R to $R - c$, and by induction, the remaining generated change by greedy will be optimal.

To show the assertion, suppose to the contrary that the optimal algorithm does not generate a coin of value c. This implies that it only use coins of values $c/3$ or smaller. These coin

denominations all evenly divide c, which implies that some subset of such coins sums to c. Thus, greedy uses a single coin c and the optimum algorithm uses at least two or more to generate this same amount c. Both systems have the same remainder $R - c$. Clearly, this contradicts the presumed optimality.

(b) (This is not the smallest counterexample, but it really happened in history.) The old British system had coins for 1 penny (1d), 3-pence (3d), 6-pence (6d), 10-pence (10d) and 1 shilling which equals 12 pence (1s = 12d). To make 20d the greedy algorithm would use four coins (a shilling, a 6-pence, and two pennies) whereas the optimum algorithm would use two coins (two 10-pence).

Solution 9: We present a dynamic programming solution. An obvious first attempt at a DP solution is to define an array $P[i]$, which is the maximum profit attainable for weeks 1 through i. The problem is that in order to update this array, we need to know where we were during the previous week since we need to know whether we need to charge the \$100 plane fare.

We will encode this extra conditional information by adding an additional parameter to control for the location where the businessman spent the last week. Let $DC = 0$ and $LA = 1$. For $0 \leq i \leq n$:

$$\begin{aligned} P[i, DC] &= \text{the max profit for weeks 1 through } i\text{, assuming week } i \text{ is spent in DC} \\ P[i, LA] &= \text{the max profit for weeks 1 through } i\text{, assuming week } i \text{ is spent in LA.} \end{aligned}$$

Let's see how to compute each of these arrays. For the basis case, because we start in DC, we pay nothing in travel costs and so we have $P[0, DC] = 0$. On the other hand, if we want to start in LA, we need to pay to get there, and thus, $P[0, LA] = -100$. (The problem states that you must start in DC, and a valid way to interpret this constraint is to forbid starting in LA, which could be done by setting $P[0, LA] = -\infty$.)

In general, for $i > 0$, to compute $P[i, DC]$, we consider two possibilities, depending on where we spent our last week. If we spent it in DC, we don't need to travel, and we obtain a profit of DC$[i]$ on top of whatever profit we accrued up to week $i-1$. Thus, we have $P[i, DC] = \text{DC}[i] + P[i-1, DC]$. On the other hand, if we were in LA last week, we need to pay the transportation costs, but we still obtain the weekly DC profit and the accrued profit from the first $i-1$ weeks. In this case we have $P[i, DC] = \text{DC}[i] + P[i-1, LA] - 100$. To maximize our profit we take the better of the two options. This yields the following recursive rule for P:

$$P[i, DC] = \text{DC}[i] + \max(P[i-1, DC], P[i-1, LA] - 100).$$

Symmetrically, to compute $P[i, LA]$, we have the rule for P:

$$P[i, LA] = \text{LA}[i] + \max(P[i-1, LA], P[i-1, DC] - 100).$$

Once we succeed in computing the values $P[i, DC]$ and $P[i, LA]$, for $0 \leq i \leq n$, we return the value $P[n, DC]$ as the final result (because we want to end in DC.) An example is shown below. The final result is $P[5, DC] = \$200$.

Week	0	1	2	3	4	5
DC		40	10	20	5	110
LA		21	90	10	150	2
$P[i, DC]$	0	40	50	70	75	200
$P[i, LA]$	-100	-79	30	40	190	192

Solution 10: Create an s-t network by making the root node r the source and creating a super-sink node t, which will have edges coming from all the terminal nodes $t_i \in T$. Observe that if $C = (X, Y)$ is any cut of the resulting s-t network, then removing the edges that cross the cut from X to Y separates r from all the terminals. Therefore, this problem is equivalent to computing a minimum weight cut in this network.

We can reduce this to a network flow problem. First, set all the capacities of all the nodes of the network to 1 except the edges that enter t to capacity ∞, or more practically, any numeric value that is larger than the maximum possible flow (see Fig. 3). Now, run any network flow algorithm on the resulting network. Let f denote the resulting flow.

We compute the minimum cut as follows. First, construct the residual graph G_f, and determine the subset of nodes X that are reachable from r, and let $Y = V \setminus X$ be the remaining nodes. Note that none of the nodes of T can be reachable from r in G_f, for otherwise we could push more flow through this terminal into t. Therefore, $T \subseteq Y$. By the max-flow/min-cut theorem, the edges (x, y) of G that cross the cut (that is, where $x \in X$ and $y \in Y$) define the minimum cut in G. Since these edges are of capacity 1, the capacity of this cut is equal to the number of edges in the cut. Therefore, this is the minimum number of edges needed to separate r from all the vertices of T.

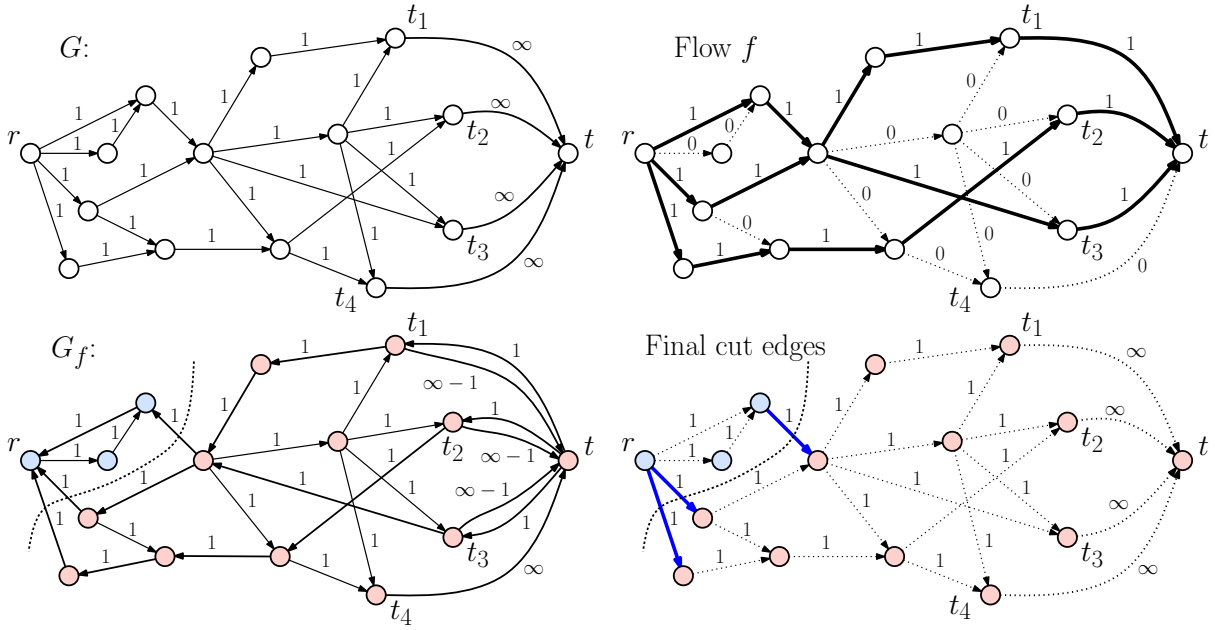

Figure 3: Eliminating edges to separated r from terminals.

Solution 11: As in class, we make the implicit assumption that there exists at least one path of strictly positive capacity from s to t. (If not, the maximum flow is zero, and no edge is critical.)

We assert that *any edge that crosses a minimum cut is critical.* To see why, suppose that (X, Y) is a minimum cut, and let (x, y) be any edge that crosses this cut for $x \in X$ and $y \in Y$. The capacity of the cut is the sum of capacities of all X-Y crossing edges, and so decreasing the capacity of (x, y) decreases the capacity of the cut. By the Min-Cut/Max-Flow Theorem, decreasing the minimum cut decreases the maximum flow value in the network.

To compute such an edge, we use the procedure suggested in the proof of the Min-Cut/Max-

Flow proof. First, compute the maximum flow f in the network (by any max-flow algorithm) and then compute the residual network G_f for this flow. Let X denote the vertices that are reachable from s in G_f, and let $Y = V \setminus X$. (Since f is maximum, there is no s-t path in G_f, implying that Y is nonempty). As shown in the min-cut/max-flow proof, (X, Y) is a cut of capacity $|f|$. Therefore, any edge (x, y) where $x \in X$ and $y \in Y$ suffices.