22C:153 Homework 2
Solutions

(1) Consider 3 matrices $A1$, $A2$ and $A3$ of orders 10×100, 100×5 and 5×2 respectively.

Using the greedy approach, the matrices should be multiplied in the order of $((A1 * A2) * A3)$ which results in $10 * 100 * 5 + 10 * 5 * 2 = 5100$ scalar multiplications.

The optimal solution is to multiply the matrices according to $(A1 * (A2 * A3))$ which results in $100 * 5 * 2 + 10 * 100 * 2 = 3000$ scalar mutliplications. So, the professor is wrong.

(2) Let W_{ij} denote the minimum weight of the triangulation of the subpolygon with vertices from i through j in considered in counter clockwise direction and d_{ij} denote the length of the line segment from between i and j.

We have to compute W_{1n}. The value of W_{ij} is given by the following formula:

$$W_{ij} = \begin{cases} 0 & \text{if } j - i \leq 3 \\ min_{i<k<j} W_{ik} + W_{kj} + d_{ik} + d_{kj} & \text{otherwise} \end{cases}$$

(3-a) Let $D((m+1) \times (n+1))$ be a 2-dimensional array, where m denotes the length of the string X and n denotes the length of the string Y. Each element in D, $d[i][j]$, $0 \leq i \leq m$, $0 \leq j \leq n$, denotes the cost of converting a string $X[1...i]$ to $Y[i...j]$.

The value of $d[i][j]$ depends on the following cases:

$$d_{ij} = \begin{cases} j * cost(insert) & \text{if } i = 0 \\ i * cost(delete) & \text{if } j = 0 \\ \min \begin{cases} d[i-1][j-1] + cost(copy) & \text{if } X[i] = Y[j] \\ d[i-1][j-1] + cost(replace) & \\ d[i-2][j-2] + cost(twiddle) & \text{if } X[i] = Y[j-1] \text{ and } X[i-1] = Y[j] \\ d[i-1][j] + cost(delete) & \\ d[i][j-1] + cost(insert) & \end{cases} & \text{otherwise} \end{cases}$$

After $d[m][n]$ is filled, its cost should be checked with cost of kill operation (which can be performed only after the last column of a row is filled). The following code can be added after the table is filled:

```
for($i \leftarrow 1$ to $m$) do
        if ($d[m][n] > d[i][n] + cost(kill)$)
                $d[m][n] = d[i]n] + cost(kill)$
```

(3-b) The operations "Copy", "Replace", "Delete" and "Insert" can be used to align both the DNA sequences. The cost of each of the following operations will be $1, -1, -2$ respectively and instead of finding the minimum cost of converting one sequence to the other, we have to find the maximum cost.

(4) For each $i = 1, 2, ..., n$, construct a $W \times W$ table T_i. Here W is the sum of all the $a_i's$. Each entry $T_i[p][q]$ is a boolean that indicates if there are disjoint subsets S_1, S_2 of $\{a_1, a_2, ..., a_i\}$ such that sum of elements in S_1 is p and sum of elements in S_2 is q. The base case is the table T_2 in which an element $T_2[p][q]$ is 1 if either $a_1 = p$ and $a_2 = q$ or $a_1 = q$ and $a_2 = p$. $T_i[p][q]$ can be filled in $O(1)$ time by noting that element a_i may be in S_1, may be in S_2, or may be in neither. This means that:

$$T_i[p][q] = \begin{cases} 1 & \text{if } T_{i-1}[p][q] = 1 \\ 1 & \text{if } T_{i-1}[p - a_i][q] = 1 \\ 1 & \text{if } T_{i-1}[p][q - a_i] = 1 \\ 0 & \text{otherwise} \end{cases}$$

Once the table T_n is filled, the optimal ratio can be found by picking the minimum ratio p/q such that $T[p][q] = 1$ and $p > q$. This can be done by simply scanning the table in $O(W^2)$ time. Once this is done, the subsets S_1 and S_2 can also be easily constructed by calling the function `constructSets`(p, q, n), where p/q is the optimal ratio.

```
constructSets($p, q, i$)
          (* Base case *)
          if ($i = 1$)
                    if ($a_1 = p$)
                              return ($\{p\}$, $\emptyset$)
                    if ($a_1 = q$)
                              return ($\emptyset, \{q\}$)
          (* Recursive case *)
          if ($T_{i-1}[p][q] = 1$)
                    return constructSets($p, q, i - 1$)
          if ($T_{i-1}[p - a_i][q] = 1$)
                    ($S_1, S_2$) $\leftarrow$ constructSets($p, q, i - 1$)
                    return ($S_1 \cup \{a_i\}, S_2$)
          if ($T_{i-1}[p][q - a_i] = 1$)
                    ($S_1, S_2$) $\leftarrow$ constructSets($p, q, i - 1$)
                    return ($S_1, S_2 \cup \{a_i\}$)
```

This recursive function takes $O(n)$ time. It takes $O(nW^2)$ time to fill all the tables. So the running time is $O(nW^2)$. This is a psuedopolynomial.

(5) Let $G(V, E)$ be the given graph. Let $d[v]$ represent the distance of vertex v from s. We will construct a graph $G'(V, E')$ in the following way:

Apply BFS to the graph G using s as the starting point, to construct distances $d[v]$ of all vertices v. This takes $O(V + E)$ time. Delete all edges u, v from G for which $|d[u] - d[v]| \neq 1$. This takes $O(E)$ time and hence total of $O(V + E)$. After this is completed, the graph G' contains only the edges which are on the shortest paths from s to t.

The next step is to apply a modified DFS algorithm to G'. Modify the DFS to run only to find one path at a time. Once a path from s to t is found or if a dead end has reached, it starts again from s. Whenever an edge is visited, remove it from the graph G and store it in a temporary graph. If this run of DFS finds a successful path, this temporary graph will be one of the shortest paths in the maximal set, else just delete the temporary graph. Each iteration of DFS takes time proportional to the number of edges that were deleted in this run Since each edge is deleted once visited, each edge is visited only once and hence this algorithm takes $O(E)$ time. Hence the total running time is $O(V + E)$. This modification gives the maximal set of shortest paths.

(6-a) Construct the graph G' as described in the hint given in the text book. Assume that each edge in the graph is of unit capacity. Apply the maximum flow algorithm to the graph G'. For each edge (x_i, y_j) selected in the maximum flow, pick the edge (i, j) in G. In this method, only one incoming edge is chosen per vertex and only one outgoing edge is choosen. Since the graph is acyclic, we obtain a path cover. After applying maximum flow algorithm to G', the edges choosen will be such that the they form paths of maximum lengths possible (leaving out very few vertices not belonging to any path of length more than 1), we get minimum path cover.

(6-b) Consider a directed 4-cycle, $1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 1$. In this case all the edges will be chosen after applying the above algorithm, thus producing a cycle instead of a path.

(7-a) Increasing capacity of an edge by 1 effects the value of maximum flow only if the flow along that edge is equal to the capacity of the edge (before increasing). In this case, the following method updates the maximum flow:

1. Construct a residual graph of the given graph with respect to the original max flow. This step takes $O(V + E)$ time.

2. Find a path from s to t in the residual graph (if there is one). This step takes $O(V + E)$ time.

3. Update the flow along this path, i.e, increase the flow on each edge of this path. This increases the maximum flow value by 1. This step takes $O(E)$ time.

This produces a maximum flow. The reason is, since the capacity is increased by only 1, the maximum value will increase by atmost 1.

(7-b) If we decrease the capacity of an edge along which the flow is less than its capacity (before decreasing), the maximum flow value is not effected. Let $\mathrm{f}(u, v)$ denote the flow along the edge (u, v) and let $c(u, v)$ denote the capacity of the edge (u, v). Let (u, v) be the edge along which the capacity is decreased. Assume that $\mathrm{f}(u, v) = c(u, v)$ (before decreasing $c(u, v)$). In this case, the following method updates the value of the maximum flow:

1. Find a path from s to u in the graph (after the capacity has been decreased), with edges containing only positive flows, and decrease the flow on each edge of this path by 1. Decrease the flow on the edge (u, v) by 1. Find a path from v to t, with edges containing only positive flows, and decrease the flow on each edge of this path by 1. Since $\mathrm{f}(u, v) > 0$, an $s - u$ path and a $v - t$ exists. Decrease the maximum flow by 1. This step can be done in $O(V + E)$.

2. Construct a residual graph of the original graph. This can be done in $O(V + E)$ time.

3. Find a path from s to t (if there is one) and increase the value of the flow on each edge of this path by 1. Then increment the maximum flow by 1. Since, the maximum flow was

decreased only by 1, executing this step only once will be sufficinet as the maximum flow can be increased by atmost 1.

(8-a) Since, the maximum capacity of an edge is C, any mincut of the graph can have atmost $C|E|$ capacity.

(8-b) Apply either DFS or BFS to the given graph, considering only those edges whose capacity is atleast k. The paths we obtain by this method have capacity atleast k. This step takes $O(V + E)$ time.

(8-c) The given algorithm functions just as the normal maximum flow algorithm. The reason is, in this algorithm, at each step of the outer while loop, all the paths of capacity atleast k are autmented and the outer loop runs till $k \geq 1$. This means that all the paths of capacity atleast 1 are selected, which is what happens in the origina maximum flow algorithm.

(8-d) For the first time the algorithm enters the outer loop, maximum capacity of an edge in the graph is C. We have,

$2k = 2 * 2^{\lfloor \lg C \rfloor} = 2^{\lfloor \lg C \rfloor + 1} \geq 2^{\lg C} = C.$

Hence, the maximum capacity of a mincut in the first step is $\leq 2k|E|$. In the successive steps, the maximum capacity of the remaining paths is less than k which means the maximum capacity of an edge in the graph is also reduced by half. Hence, the maximum capacity of a mincut is atmost $2k|E|$.

(8-e) Since, in the inner loop, the capacity of a path is atleast k and the maximum capacity is atmost $2k|E|$, the inner while loop runs atmost $O(|E|)$ times.

(8-f) The outer most while loop runs atmost $\lg C$ times. The inner while loop runs $O|E|$ times and each execution of the inner while loop takes $O|E|$ time. Hence the algorithm takes $O(\lg CE^2)$ times.

(9-a) $a_0 = R^0 = 1$. Hence the given statement is true for 0.

Assume that the statement is true for $n - 1$. To prove for n.

$$\begin{aligned} a_n = a_{n-2} - a_{n-1} &= R^{n-2} - R^{n-1} \\ &= R^{n-2}(1 - R) \\ &= R^{n-2}a_2 \\ &= R^{n-2}R^2 = R^n. \end{aligned}$$

(9-b) After the first augmentation (given in the problem), augment the following four paths in order:

$s \to a \to b \to c \to d \to t$. Send a flow of R along this path.
$s \to c \to b \to a \to t$. Send a flow of R along this path.
$s \to a \to b \to c \to d \to t$. Send a flow of $1 - R$ along this path.
$s \to d \to c \to b \to t$. Send a flow of $1 - R$ along this path.

After the above four augmentations, we get the desired residual capacities.

(9-c) Consider the above sequence of four augmentation paths as a round. Repeat augmenting the paths in this round. After n rounds i.e, after augmenting the four paths in the above sequence for n times, the residual capacities on edges (c,d), (a,b), (b,c) are a_{2n}, a_{2n+1} and 1 respectively.

The value of flow at this point is:

$2(a_1 + a_2 + a_3 + ... + a_{2n-1} + a_{2n}) + 1$

$= 2(R^1 + R^2 + ... + R^{2n-1} + R^{2n}) + 1$

$= 2\frac{R(1-R^n)}{1-R} + 1$

As $n \to \infty$ the value of the flow is $\frac{2R}{1-R} + 1 = 2 + \sqrt{5}$

In a round, of the edges with original capacity M, edges $s \to a$ and $d \to t$ are used twice and other edges are used only once. Let us consider the flow along the edge $s \to a$. In the first round, flows of R and $1 - R$ are sent through this edge. So, through all the rounds, the flow along this edge is half the total flow which is:

$1 + \sqrt{\frac{5}{2}} = 3$ (rounded off to the next higher integer).

There is a flow of 1 along this edge at the beginning. Hence, the augmentation along the paths in a round can be continued infinitely many times without worrying about saturarting the edges with original capacity of M.

(10) The following algorithm finds the edge connectivity of a graph. Before using the algorithm below, replace each edge $(u,v) \in E$ in the given graph with two directed edges (u,v) and (v,u).

```
$s = u \in V$
for each $v \in V - u$
    $t = v$
    maximum-flow(G,s,t)
return the minimum cut of all the above maximum flow function calls.
```

This algorithm finds the edge-connectivity for the following reason. Let C be a min-cut of G and let t be a vertex that is not in the same connected component as s in $G - C$. THen when the above algorithm considers this (s,t) pair, the max flow it finds will be $|C|$. It is also immediate that if in some iteration the algorithm finds a max-flow with value less than $|C|$, then there is a cut of size less than $|C|$, contradiction. Hence, the algorithm correctly identifies $|C|$ as the edge connectivity of G.