CS 170 HW 7

Due **2020-10-19, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Egg Drop Revisited

Recall the Egg Drop problem from Homework 6:

You are given k identical eggs and an n story building. You need to figure out the highest floor $\ell \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor ℓ or lower, and always breaks if dropped from floor $\ell+1$ or higher. ($\ell = 0$ means the egg always breaks). Once an egg breaks, you cannot use it any more.

Let $f(n, k)$ be the minimum number of egg drops that are needed to find ℓ (regardless of the value of ℓ).

Instead of solving for $f(n, k)$ directly, we define a new subproblem $M(d, k)$ to be the maximum number of floors for which we can always find ℓ in at most d drops using k eggs.

(a) Find a recurrence relation for $M(d, k)$ that can be computed in constant time given the previous subproblems. Briefly justify your recurrence.

(Hint: *As a starting point, what is the highest floor that we can drop the first egg from and still be guaranteed to solve the problem with the remaining $d-1$ drops and $k-1$ eggs if the egg breaks?*)

(b) Give an algorithm to compute $M(d, k)$ given d and k and analyze its runtime.

(c) Modify your algorithm from (b) to compute $f(n, k)$ given n and k. (Hint: *If we can find ℓ when there are more than n floors, we can also find ℓ when there are n floors.*)

(d) Show that the runtime of the algorithm of part (c) is $O(nk)$.

(e) How can we implement the algorithm using $O(k)$ space?

Solution:

(a)

$$M(d, k) = M(d-1, k-1) + M(d-1, k) + 1$$

Consider any strategy that uses d drops and k eggs drops the first egg from floor i. There are two outcomes: If the egg breaks, we have reduced the problem to floors 1 to $i-1$, and the strategy can solve this subproblem using $d-1$ drops and $k-1$ eggs. The optimal strategy for $d-1$ drops and $k-1$ eggs can distinguish between $M(d-1,k-1)$ floors, so we should choose $i = M(d-1,k-1)+1$.

On the other hand, if the egg doesn't break, we reduce the problem to floors $i+1$ to n with $d-1$ drops and k eggs. The maximum number of floors we can distinguish using this many drops and eggs is $M(d-1,k)$, so we can solve this subproblem for n as large as $M(d-1,k)+i = M(d-1,k-1)+M(d-1,k)+1$.

(b) For base cases, we'll take $M(0,k)=0$ for any k and $M(d,0)=0$ for any d. Starting with $d=1$, we compute $M(d,x)$ for all, $1 \le x \le k$, and do so again for increasing values of d, up until we compute $M(d,x)$ for all $1 \le x \le k$. We return $M(d,k)$.

We compute dk subproblems, each of which takes constant time, so the overall runtime if $\Theta(dk)$.

(c) Again, starting with $d=1$, we compute $M(d,x)$ for all, $1 \le x \le k$, and do so for increasing values of d. This time, we stop the first time we find that $M(d,k) \ge n$, and return this value of d.

(d) Because there are only n floors, the optimal number of drops, d will always be at most n. From part (b), we know the runtime is $\Theta(dk)$, so if $d \le n$, we know the runtime must be $O(nk)$.

(This is a very loose bound, but it is still much better than the naive $O(n^2k)$-time algorithm we'd get from using the recurrence on $f(n,k)$ directly.)

(e) While we're computing $M(d,x)$ for all $1 \le x \le k$, we only need to store $M(d-1,x)$ and $M(d,x)$ for all x, i.e. we only ever need to store $O(k)$ values. In particular, after computing $M(d,x)$ for all x, we can delete our stored values of $M(d-1,x)$.

3 Maximum Non-Crossing Bipartite Matching

Recall that given a graph G, a matching in G is a set of edges in G such that no two edges share an endpoint.

We are given a bipartite graph $G(L \cup R, E)$, where L and R are ordered. That is, the vertices of G can be partitioned into two ordered sets $L = \{\ell_1, \ell_2, \cdots \ell_m\}$ and $R = \{r_1, r_2 \dots r_n\}$, such that every edge is of the form (ℓ_i, r_j).

We say a matching in G is *non-crossing* if it does not contain two edges $(\ell_i, r_j), (\ell_{i'}, r_{j'})$ such that $i < i'$ but $j > j'$. Pictorially, think of ℓ_i as being located at the point $(0,i)$ and r_j as being located at the point $(1,j)$. Then a matching is non-crossing if we can draw the line segment between the endpoints of every edge in the matching, without any of these line segments intersecting.

(a) Show that in any non-crossing matching in G, at least one of the following is true: (i) the edge (ℓ_m, r_n) exists and is in the matching, (ii) ℓ_m is not included in the matching, or (iii) r_n is not included in the matching.

(b) Give an $O(mn)$-time dynamic programming algorithm to find the size of largest non-crossing matching in G. Give a three-part solution. (Hint: Part (a) identifies cases that might be useful in writing the recurrence relation).

Solution:

(a) If both of ℓ_m, r_n are matched but not using the edge (ℓ_m, r_n), then ℓ_m is matched to some r_j where $j < n$ and r_n is matched to some ℓ_i where $i < m$. However, the edges (ℓ_m, r_j) and (ℓ_i, r_n) both being in the matching violates the non-crossing condition, a contradiction.

(b) **Main idea:** Let $M(i, j)$ be the size of the largest non-crossing matching using only vertices ℓ_1 to ℓ_i and r_1 to r_j. Our base cases are $M(i, 0) = 0$ and $M(0, j)$ for all i, j.

The recurrence is:

$$M(i,j) = \max \begin{cases} 1 + M(i-1, j-1) & \text{if } (\ell_i, r_j) \in E \\ M(i-1, j) \\ M(i, j-1) \end{cases}$$

Our algorithm computes all $M(i, j)$ (e.g. in increasing order of i then j), and outputs $M(m, n)$.

Proof of correctness

Our base cases are correct, since if one of L, R is empty any matching has size 0.

Inductively assume we computed all previous values of $M(\cdot, \cdot)$ correctly before computing $M(i, j)$. By part (a), there are three cases to consider for the maximum non-crossing matching between vertices ℓ_1 to ℓ_i and r_1 to r_j:

- The edge (ℓ_i, r_j) exists and is in the matching. In this case, deleting these vertices and this edge gives the maximum non-crossing matching on vertices ℓ_1 to ℓ_{i-1} and r_1 to r_{j-1}, i.e. $M(i, j) = 1 + M(i-1, j-1)$.
- ℓ_i is not in the matching. In this case, deleting ℓ_i doesn't affect the size of the matching, i.e. $M(i, j) = M(i-1, j)$.
- r_j is not in the matching. Similar to the previous case, $M(i, j) = M(i, j-1)$.

Our algorithm computes the size of the maximum matching in each case that applies and then outputs the largest of these, so $M(i, j)$ must be computed correctly.

Runtime analysis There are mn subproblems, and each one takes $O(1)$ time to solve, so the runtime is $O(mn)$.

Comment: Although this may not look like a string problem, this is actually a generalization of the longest common subsequence problem, where we are given two strings and want to find the longest string that is a subsequence of both. If we think of the characters in each string as vertices, and there is an edge between two vertices if the corresponding characters match, then any subsequence of both strings corresponds to a non-crossing matching. The longest common subsequence problem is solved often in computational inguistics, data comparison, and computational biology.

4 Jeweler

This is a solo question.

You are a jeweler who sells necklaces and rings. Each necklace takes 4 ounces of gold and 2 diamonds to produce, each ring takes 1 ounce of gold and 3 diamonds to produce. You have 80 ounces of gold and 90 diamonds. You make a profit of 60 dollars per necklace you sell and 30 dollars per ring you sell, and want to figure out how many necklaces and rings to produce to maximize your profits.

(a) Formulate this problem as a linear programming problem. Draw the feasible region, and find the solution (state the cost function, linear constraints, and all vertices except for the origin).

(b) Suppose instead that the profit per necklace is C dollars and the profit per ring remains at 30 dollars. For each vertex you listed in the previous part, give the range of C values for which that vertex is the optimal solution.

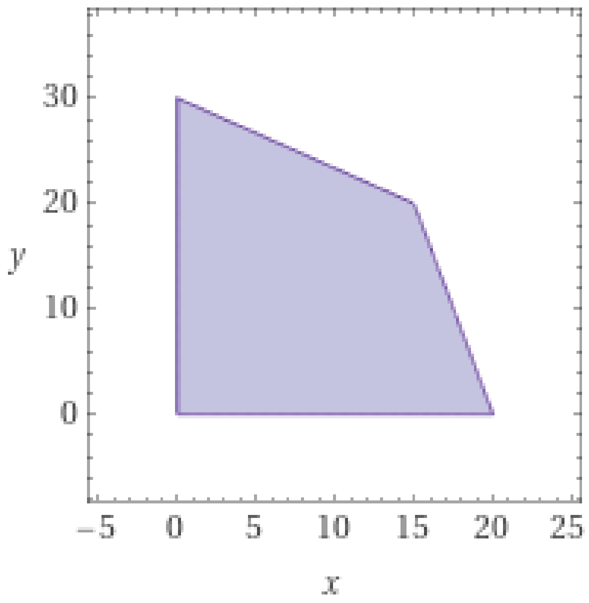
Solution:

(a) x = number of necklaces
y = number of engagement rings

Maximize: $60x + 30y$

Linear Constraints:
$4x + y \leq 80$
$2x + 3y \leq 90$
$x \geq 0$
$y \geq 0$

The feasible region:



The vertices are $(x = 20, y = 0), (x = 15, y = 20), (x = 0, y = 30)$, and the objective is maximized at $(x = 15, y = 20)$, where $60x + 30y = 1500$.

(b) There are lots of ways to solve this part. The most straightforward is to write and solve a system of inequalities checking when the objective of one vertex is at least as large as the objective of the other vertices. For example, for $(x = 15, y = 20)$ the system of inequalities would be $C \cdot 15 + 30 \cdot 20 \geq C \cdot 20$ and $C \cdot 15 + 30 \cdot 20 \geq 30 \cdot 30$. Doing this for each vertex gives the following solution:

$(x = 0, y = 30) : C \leq 20$
$(x = 15, y = 20) : 20 \leq C \leq 120$
$(x = 20, y = 0) : 120 \leq C$

One should note that there is a nice geometric interpretation for this solution: Looking at the graph of the feasible region, as C increases, the vector $(C, 30)$ starts pointing closer to the x-axis. The objective says to find the point furthest in the direction of this vector, so the optimal solution also moves closer to the x-axis as C increases. When $C = 20$ or $C = 120$, the vector $(C, 30)$ is perpendicular to one of the constraints, and there are multiple optimal solutions all lying on that constraint, which are all equally far in the direction $(C, 30)$.

5 Modeling: Tricks of the Trade

One of the most important problems in the field of *statistics* is the *linear regression problem.* Roughly speaking, this problem involves fitting a straight line to statistical data represented by points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ on a graph. Denoting the line by $y = a + bx$, the objective is to choose the constants a and b to provide the "best" fit according to some criterion. The criterion usually used is the *method of least squares*, but there are other interesting criteria where linear programming can be used to solve for the optimal values of a and b.

Suppose instead we wish to minimize the sum of the absolute deviations of the data from the line, that is,

$$\min \sum_{i=1}^{n} |y_i - (a + bx_i)|$$

Write a linear program with variables a, b to solve this problem.
Hint: Create a new variable z_i that will equal $|y_i - (a + bx_i)|$ in the optimal solution.

Solution:
Note that the smallest value of z that satisfies $z \geq x, z \geq -x$ is $|x|$. Hence

$$\min \sum_{i=1}^{n} |y_i - (a + bx_i)|$$

is equivalent to

$$\text{Minimize } \sum_{i=1}^{n} z_i$$
$$\text{subject to } \begin{cases} y_i - (a + bx_i) \leq z_i & \text{for } 1 \leq i \leq n \\ (a + bx_i) - y_i \leq z_i & \text{for } 1 \leq i \leq n \end{cases}.$$

Since the optimal solution will set $z_i = |y_i - (a + bx_i)|$

6 Max Flow with Multiple Sinks/Sources

This is a solo question.

Consider the following variation of max flow: You are given a directed graph G, with capacities on all the edges. There are 2 source vertices s_1, s_2 and 2 sink vertices t_1, t_2, and now flow is allowed to originate from either source and arrive at either sink.

Call any flow that satisfies the usual capacity constraints on all edges and balance constraints for all non-source/sink vertices a 2-flow. Informally, a 2-flow is just like a flow, except it can originate at either source and move to either sink. The size of a 2-flow is the total amount of flow leaving s_1 and s_2 (equivalently, the total amount arriving at t_1 and t_2).

Show how to construct a directed graph G' (with edge capacities and one designated source and sink vertex) such that:

(1) If F is a 2-flow in G, there is a flow F' in G' of the same size,

(2) If F' is a flow in G', then there is a 2-flow F in G of the same size.

In addition to describing how to construct G', describe how to find F' given F and vice-versa.

Solution: Solution 1: To construct G', we take G and add two new vertices s', t'. For both s_i, we add an edge (s', s_i) with capacity ∞. For both t_j, we add an edge (t_j, t') with capacity ∞. s' will be the source in G', and t' will be the sink. The idea is that since s' has infinite capacity edges to both s_i, it is possible to mimic originating any amount of flow at s_i by just originating the same amount of flow at s' and pushing it through the edge (s', s_i) (and the same is true for mimicing sinking flow at t_j).

Given any 2-flow F in G, we can construct a flow F' in G' of the same size as follows: We use the same flow values for all edges in G, and then use the flow value $\sum_{(s_i,v)\in E} f(s_i, v)$ for each edge (s, s_i), and similarly use the flow value $\sum_{(v,t_j)\in E} f(v, t_j)$ for each edge (t_j, t). Since the new edges have infinite capacity, capacity constraints remain satisfied, and conservations constraints are satisfied for all s_i, t_j by construction.

Given any flow F' in G', we can construct a 2-flow F in G of the same size as follows: simply use the same flow values from F' for all edges in G (and ignore the flow values on the edges only appearing in G') to get a valid 2-flow.

Solution 2: To construct G' from G, we delete both s_i (and all adjacent edges), and add a new vertex s'. s' is the source vertex in G'. For every edge (s_i, v) that was deleted, we add a new edge (s', v) in G' with the same capacity. We will allow there to be parallel edges in G' if both s_i had an edge to the same vertex in G (we could always replace the parallel edges with a single edge whose capacity is the sum of the parallel edges' capacities if parallel edges are undesirable). Next, we delete both t_j, add a new vertex t' that is the sink vertex, and for every edge (v, t_j) that was deleted, we add a new edge (v, t') with the same capacity. (Note that if there is an edge of the form (s_i, t_j), it will eventually be replaced with an edge from (s', t') in this process).

The idea is that since we don't care which source flow leaves from, and sending flow between the sources doesn't matter, we can just combine all sources without affecting the set of feasible flows. A similar statement holds for the sinks.

Given any 2-flow F in G, we can construct a flow F' in G' of the same size as follows: For every edge (s_i, v) or (v, t_j), assign the same flow value to the corresponding edge (s', v) or (v, t') in G'. For every other edge in G, it still exists in G', so assign the same edge in G' the same flow value.

Given any flow F' in G', we can construct a 2-flow F in G of the same size as follows: For every edge (s', v) or (v, t'), assign the same flow value to the corresponding edge (s_i, v) or (v, t_j) in G'. For every other edge in G', it exists in G, so assign the same edge in G the same flow value.

7 Routing Data

The internet is modelled as a directed network $G = (V, E)$, where the vertices are data centers, and edges represent connections between data centers. There are k types of data, and for the ith type of data, there is a source data center s_i, a destination data center t_i, and we want to transfer at least r_i units of this type of data through the network from s_i to t_i (we are allowed to transfer more). Using edge $e = (u, v)$, we can transfer at most c_e total units of data from u to v. For example, if $c_e = 2$, we could use e to transfer 2 units of type 1 data, or 1 unit of each of type 1 and type 2 data. Our goal is to come up with a plan for routing each type of data, so that the total amount of data transferred is maximized.

Let $f_{i,e} \geq 0$ denote the number of units of the ith type of data we transfer using edge e. Describe how to write each of the following aspects of this problem as linear programming constraints or objectives in terms of these variables.

(a) For any vertex besides s_i, t_i, the amount of data type i arriving at that vertex is the same as the amount leaving.

Solution:

$$\forall i, v \neq s_i, t_i : \sum_{e=(u,v)\in E} f_{i,e} = \sum_{e=(v,u)\in E} f_{i,e}$$

(b) The total amount of any kind of data being transfered through edge e is at most c_e units.

Solution:

$$\forall e : \sum_i f_{i,e} \leq c_e$$

(c) At least r_i units of data type i leave s_i (and arrive at t_i).

Solution:

$$\forall i : \sum_{e=(s_i,v)} f_{i,e} \geq r_i$$

$$\forall i : \sum_{e=(v,t_i)} f_{i,e} \geq r_i$$

We can actually drop the constraint t_i, since the constraint for t_i and the first set of constraints guarantee that $\sum_{e=(s_i,v)} f_{i,e} = \sum_{e=(v,t_i)} f_{i,e}$

(d) The total amount of data being transferred (i.e., the sum over the source data centers of the amount of data leaving them) is as large as possible.

Solution:

$$\max \sum_i \sum_{e=(s_i,v)} f_{i,e}$$

8 (Extra Credit) Knightmare

Give an efficient algorithm to find the number of ways you can place knights on an N by M ($M < N$) chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). Clearly describe your algorithm and prove its correctness. Your algorithm's runtime can be exponential in M but should be polynomial in N.

(Please provide a 3-part solution)

Solution:

We use length M bit strings to represent the configuration of rows of the chessboard (1 means there is a knight in the corresponding square and otherwise 0).

The main idea of the algorithm is as follows: we solve the subproblem of the number of valid configurations of $(n-1) \times M$ chessboard and use it to solve the $n \times M$ case. Note that as we iteratively incrementing n, a knight in the n-th rows can only affect configurations of rows $n+1$ and $n+2$. So we can denote $K(n, u, v)$ as the number of possible configurations of the first n rows with u being the $(n-1)$-th row and v being the n-th row, and then use dynamic programming to solve this problem.

Let a list of bitstrings be valid if placing the knights in the first row according to the first bitstring, in the second row according to the second bitstring, etc. doesn't cause two knights to attack each other. Then we have $K(2, u, v) = 1$ if u, v are valid and 0 otherwise for all u, v pairs.

For $K(n, v, w)$ we have:

$$K(n, v, w) = \sum_{u : u,v,w \text{ are valid}} K(n-1, u, v)$$

Proof of Correctness: The only 2-row configuration of knights ending in row configurations u, v is the configuration u, v itself. So $K(2, v, w) = 1$ if u, v are valid. Otherwise, $K(2, v, w) = 0$.

For $n > 2$, the above recurrence is correct because for any valid n-row configuration ending in v, w, the first $n-1$ rows must be a valid configuration ending in u, v for some u, and for this same u, the last three rows u, v, w must be also valid configuration. Moreover, this correspondence between n-row and $(n-1)$-row configurations is bijective.

Runtime Analysis: To bound the total runtime, first note that for each row, we iterate over all possible configurations of knights in the row and for each such configuration, we perform a sum over all possible valid configurations of the previous two rows. The time to check if a configuration is valid is $O(M)$. Therefore, the time taken to compute the subproblems for a single row is $O(2^{3M}M)$ which gives us an overall runtime of $O(2^{3M}MN)$.