CS 170 HW 9 (Optional)

Due **2021-11-01, 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"

You may submit your solutions if you wish them to be graded, but they will be worth no points

The two problems below are dynamic programming problems and should be viewed as having 3 parts each. You should find a function f which can be computed recursively so that evaluation of f on a certain input (or combining its evaluations on a few inputs) gives the answer to the stated problem.

- Part (a) is to define f *in words* (without mention of how to compute it recursively). You should clearly state how many parameters f has, what those parameters represent, what f evaluated on those parameters represents, and how you should use f to get the answer to the stated problem.
- Part (b) is to give a recurrence relation showing how to compute f recursively, including a description of the base cases.
- In part (c) you should give the running time **and space** for solving the original problem using computation of f via memoization or bottom-up dynamic programming. If you need to use certain data structures to make computation of f faster, you should say so. **Note:** if there are multiple solutions to solve the stated dynamic programming problem, you should describe the most time-efficient one you know. If there are multiple solutions with the same asymptotic time complexity, you should describe the implementation that gives the best asymptotic space complexity.

2 Equivalent Strings

We are given two strings A, B of length n, m respectively. These two strings can contain English characters a to z, as well the special character ?. We say A and B are equivalent if it is possible to replace every instance of ? with a (possibly empty) string of English characters, such that the resulting strings (containing only English letters) are the exact same.

For example, "ab?" is equivalent to "a?cd", since with the above replacements we can transform both strings into "abcd". Similarly, "a?bc" is equivalent to "abc", since we are allowed to replace ? with the empty string.

Give an efficient dynamic programming algorithm to determine if two strings are equivalent. Give a three-part DP solution as defined above.

Solution:

Main idea: Let $E(i, j)$ be True if the first i characters of A and j characters of B are equivalent, and False otherwise.

As a base case, $E(0, 0) =$ True and $E(i, 0) =$ True if and only if the first i characters of A are ? for all $i > 0$. We initialize all $E(0, j)$ analogously.

For $E(i, j)$ where $i, j > 0$ we set it to be the OR of the following cases:

1. $E(i-1, j-1)$ AND $(A[i] = B[i])$
2. $(E(i, j-1)$ OR $E(i-1, j))$ AND $(A[i] =?$ OR $B[j] =?)$

Correctness: For the base cases, two empty strings are of course equivalent, and the only non-empty strings that an empty string can be equivalent to are those consisting only of ? characters.

For the remaining values, $A[1 : i]$ and $B[1 : j]$ are equivalent if and only if one of the following possibilities happens:

- The last character of $A[1 : i]$ is matched to the last character of $B[1 : j]$ and vice-versa, and $A[1 : i-1], B[1 : j-1]$ are also equivalent.
- The last character of $A[i]$ is ?. In this case, if $A[1 : i-1]$ and $B[1 : j]$ are equivalent, then so are $A[1 : i]$ and $B[1 : j]$ since we can replace ? with the empty string. The same is true if $B[j] =?$ and $A[1 : i]$ and $B[1 : j-1]$ are equivalent.
- Alternatively, if $A[i]$ is ? and $A[1 : i]$ and $B[1 : j-1]$ are equivalent, then so are $A[1 : i]$ and $B[1 : j]$, since we can append $B[j]$ to whatever substring we replace ? with in $A[1 : i]$. The same is true if $B[j] =?$ and $A[1 : i-1]$ and $B[1 : j]$ are equivalent.

The first possibility is covered by our first case, and the second and third possibility are covered by our second case.

Runtime analysis: There are $O(nm)$ values of $E(i, j)$ to compute, each takes $O(1)$ time, so the runtime is $O(nm)$.

Partial credit $O(nm(n+m))^2$-time algorithm:

A slower but still correct solution is to use the same $E(i, j)$ and base cases as before, and use the following recurrence relation: $E(i, j)$ is the OR of:

For $E(i, j)$ where $i, j > 0$ we set it to be the OR of the following cases:

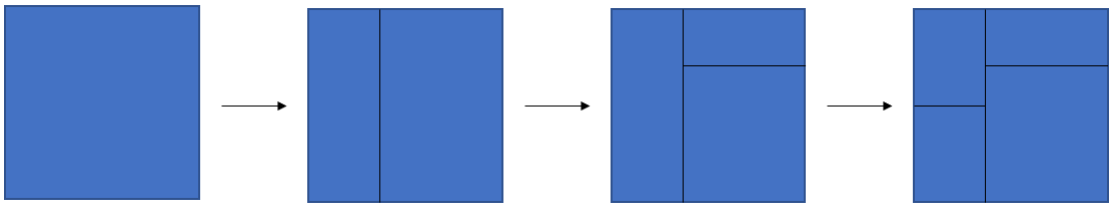
1. $E(i-1, j-1)$ AND $(A[i] = B[i])$
2. There exists k such that $E(i-1, j-k)$ AND $A[i] =?$.
3. There exists k such that $E(i-k, j-1)$ AND $B[j] =?$.

This takes $O(nm(n+m))$ time since there are potentially $n+m$ subproblems that need to be checked to compute each $E(i, j)$.

3 Geometric Knapsack

Suppose we have a piece of cloth with side lengths X, Y, where X, Y are positive integers, and a set of n *products* we can make out of the cloth. Each product is a rectangle of dimensions $a_i \times b_i$ and of value c_i, where all these numbers are positive integers.

We want to cut our large piece of cloth into multiple smaller rectangles to sell as products. Any rectangle not matching the dimensions of one of these products gets us no value. To cut the cloth, we are using a machine that takes one piece of cloth and cuts it into two, where the dividing line between the two pieces must be a vertical or horizontal line going all the way through the cloth. For example, the following is a valid series of cuts:



Give an efficient algorithm that determines the maximum value of the products that can be made out of the single $X \times Y$ piece of cloth. You may produce a product multiple times, or none if you wish. Give a three-part DP solution as defined above.

Solution:

Main idea:

For $1 \le i \le X$ and $1 \le j \le Y$, let $C(i, j)$ be the best return that can be obtained from a cloth of shape $i \times j$. Define also a function `rect` as follows:

$$\texttt{rect}(i,j) = \begin{cases} \max_k c_k & \text{over all products } k \text{ with } a_k = i \text{ and } b_k = j \\ 0 & \text{if no such product exists} \end{cases}$$

Then the recurrence relation is

$$C(i,j) = \max\left\{ \max_{1\le k<i} \{C(k,j) + C(i-k,j)\}, \max_{1\le h<j} \{C(i,h) + C(i,j-h)\}, \texttt{rect}(i,j) \right\}$$

Effectively, we look at every place we can cut the cloth and compute the resulting value, and also look at the value if we sell the cloth as one piece. We then take the max over these options. We use the convention here that if $i = 1$, $\max_{1\le k<1}\{C(k,j) + C(i-k,j)\} = 0$ (and similarly for j).

The base case is $C(1,1) = \texttt{rect}(1,1)$. The final solution is then the value of $C(X,Y)$.

To implement the algorithm, we simply initialize a $X \times Y$ 2D array. Then we fill the DP table row by row from the top, starting from $C(1,1)$ and applying the recurrence relation for each entry.

Correctness:

For proving correctness, notice that $C(1,1)$ of course is the max value we can retrieve from a 1-by-1 piece of cloth, since all products have positive integer side lengths. Inductively, $C(i,j)$ is solved correctly, as a rectangle $i \times j$ can only be cut in the $(i-1)+(j-1)$ ways considered by the recursion or be occupied completely by a product, which is accounted for by the $\texttt{rect}(i,j)$ term.

Runtime analysis:

The running time is $O(XY(X+Y+n))$ as there are XY subproblems and each takes $O(X+Y+n)$ to evaluate. This can be improved to $O(XY(X+Y)+n)$ by precomputing `rect`, but this is not necessary for full credit.

4 Vertex Cover Dual

In this problem, we consider the unweighted vertex cover problem. In this problem, we are given a graph and want to find the smallest set of vertices S such that every edge has at least one endpoint in S.

Recall the LP for this problem:

$$\min \sum_v x_v \text{s. t. } \forall (u,v) \in E : x_u + x_v \geq 1, \forall v \in V : x_v \geq 0$$

In an integral solution, $x_v = 1$ if we include v in our vertex cover, and $x_v = 0$ if we don't include v.

(i) Write the dual of the vertex cover LP. Your dual LP should have a variable y_e for every edge.

(ii) Consider the integer version of the dual you wrote, i.e. we enforce $y_e \in \{0,1\}$. Similarly to vertex cover, we can interpret $y_e = 1$ as indicating that we include e in our solution and $y_e = 0$ if we don't include e.

Using this interpretation, what does the objective say? What do the constraints say? What problem is this? (You don't have to be formal.)

(iii) True or False: If we have an integer primal solution with cost C and a fractional dual solution with cost at least $C/2$, the size of the vertex cover corresponding to the primal solution is at most twice the size of the smallest vertex cover. Briefly justify your answer.

Solution:

(i) Let E_v all the edges incident on vertex v. The dual LP is given by

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} y_e \\ \text{subject to} \quad & \sum_{e \in E_v} y_e \leq 1 \quad \forall v \in V \\ & y_e \geq 0 \qquad \forall e \in E \end{aligned}$$

(ii) The objective is to maximize the number of edges in our solution.

The constraint says no two edges in our solution share an endpoint.

This is exactly the maximum matching problem. (You don't need to name the maximum matching problem for full credit).

(iii) True. The size of the vertex cover corresponding to the integer primal solution is C. Let P_{OPT} be the optimal value of the primal LP, D_{OPT} be the optimal value of the dual LP, and V_{OPT} be the size of the smallest vertex cover. $C/2 \leq D_{OPT}$ since the dual optimal solution's objective is at least that of any feasible dual solution, $D_{OPT} = P_{OPT}$ by duality, and $P_{OPT} \leq V_{OPT}$ because the smallest vertex cover gives a feasible solution to the vertex cover LP with cost V_{OPT}. Putting it all together, we have $C \leq 2V_{OPT}$.

5 Permutation Games

A permutation game is a special form of zero-sum game. In a permutation game, the payoff matrix is n-by-n, and has the following property: Every row and column contains exactly the entries $p_1, p_2, \ldots p_n$ in some order. For example, the payoff matrix might look like:

$$P = \begin{bmatrix} p_1 & p_2 & p_3 \\ p_2 & p_3 & p_1 \\ p_3 & p_1 & p_2 \end{bmatrix}$$

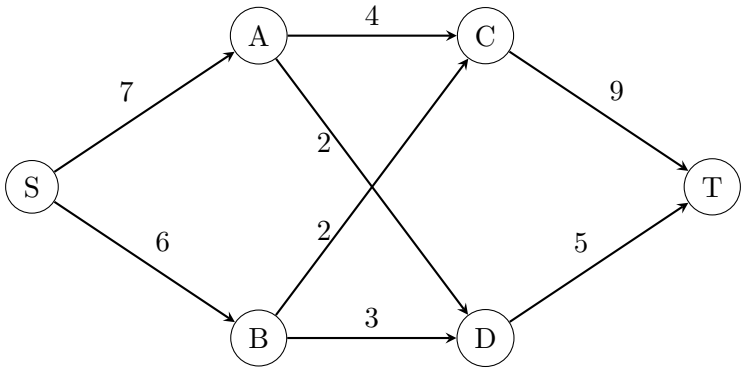
Given an arbitrary permutation game, describe the row and column players' optimal strategies, justify why these are the optimal strategies, and state the row player's expected payoff (that is, the expected value of the entry chosen by the row and column player).

Solution: Both players' optimal strategy is to choose a row/column uniformly at random. The expected payoff of this strategy is $\sum_k p_k/n$.

To show these are optimal, note that if the column player picks this strategy, any strategy the row player chooses has expected payoff $\sum_k p_k/n$. By symmetry, this also implies that if the row player picks this strategy, any strategy the column player picks achieves the same expected payoff. By duality, this must be the optimal pair of mixed equilibrium strategies.

6 Bottleneck Edges

Consider the following network (the numbers are edge capacities):



(a) Find the following:

- A maximum flow f, specified as a list of $s-t$ paths and the amount of flow being pushed through each.

- A minimum cut (the set of edges with the smallest total capacity, whose removal disconnects S and T), specified as a list of edges that are part of the cut.

(b) Draw the residual graph G_f (along with its edge capacities). In this residual network, mark the vertices reachable from S and the vertices from which T is reachable.

(c) An edge of a network is called a *bottleneck edge* if increasing its capacity results in an increase in the maximum flow. List all bottleneck edges in the above network.

(d) Give a very simple example (containing at most four nodes) of a network which has no bottleneck edges.

(e) Give an efficient algorithm to identify all bottleneck edges in a network. (Hint: Start by running the usual network flow algorithm, and then examine the residual graph.)

Solution:

(a) The maximum flow is given by the following sequence of updates:

- Route 4 units of flow along $S - A - C - T$
- Route 2 units along $S - A - D - T$
- Route 2 units along $S - B - C - T$
- Route 3 units along $S - B - D - T$

The resulting flow is feasible and has value 11. It produces the mincut $(\{S, A, B\}, \{C, D, T\})$ (the edges across the cut are (A, C), (A, D), (B, C), (B, D)) of capacity 11, certifying the optimality of the flow.

Notice that $(\{S, A, B, D\}, \{C, T\})$ (edges (A, C), (B, C), (D, T)) is also a mincut.

(b) The residual graph is shown in the figure. Vertices S,A and B are reachable from S. T can be reached from vertices C and T.

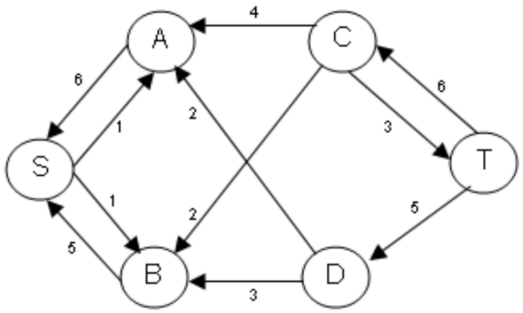


Figure 1: The residual graph

(c) (A, C) and (B, C) are bottleneck edges. The other edges belonging to a mincut are not bottleneck, as increasing their capacity does not increase the capacity of the minimum (s, t)-cut.

(d) The following figure shows an example with no bottleneck edges. The optimal flow saturates all edges, but augmenting the capacity of any of them does not increase the capacity of the minimum cut, i.e. does not open up any new path for flow to run from source to sink.

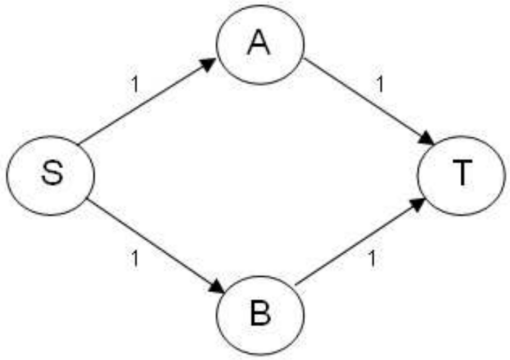


Figure 2: An example with no bottleneck edges

(e) Run the usual network flow algorithm and consider the final residual graph. Let S be the set of vertices reachable from s and T the set of vertices from which t is reachable in this graph. By the optimality of the flow S and T must be disjoint, as they are separated by a saturated cut. Suppose now that $e = (u, v)$ is a bottleneck edge. As we increase e's capacity we are able to route flow from s to u, through e and from v to t in the residual graph. This implies that $u \in S$ and $v \in T$. Moreover, if an edge $e = (u, v)$ has $u \in S$ and $v \in T$ (notice that e must then be in a minimum cut), increasing its capacity allows us to route flow from s to u (as $u \in S$), through the new capacity of e and to t (as $v \in T$). Hence, the set of bottleneck edges is the set $E(S, T)$, i.e. the set of edges which originate in S and end in T.

The running time is dominated by computing the max flow. Using the Ford-Fulkerson algorithm from the lecture and textbook, this is $O(|E| \cdot f)$, where f is the size of the max flow.

7 Multiplicative Weights Intro

Multiplicative Weights

This is an online algorithm, in which you take into account the advice of n experts. Every day you get more information on how good every expert is until the last day T.

Let's first define some terminology:

- $x_i^{(t)}$ = proportion that you 'trust' expert i on day t
- $l_i^{(t)}$ = loss you would incur on day t if you invested everything into expert i

- total regret: $R_T = \sum_{t=1}^{T} \sum_{i=1}^{n} x_i^{(t)} l_i^{(t)} - \min_{i=1,\ldots,n} \sum_{t=1}^{T} l_i^{(t)}$

$\forall i \in [1, n]$ and $\forall t \in [1, T]$, the multiplicative update is as follows:

$$w_i^{(0)} = 1$$

$$w_i^{(t)} = w_i^{(t-1)}(1-\epsilon)^{l_i^{(t-1)}}$$

$$x_i^{(t)} = \frac{w_i^{(t)}}{\sum_{i=1}^{n} w_i^{(t)}}$$

If $\epsilon \in (0, 1/2]$, and $l_i^{(t)} \in [0, 1]$, we get the following bound on total regret:

$$R_T \leq \epsilon T + \frac{\ln(n)}{\epsilon}$$

Let's play around with some of these questions. For this problem, we will be running the randomized multiplicative weights algorithm with two experts. Consider every subpart of this problem distinct from the others.

(a) Let's say we believe the best expert will have cost 20, we run the algorithm for 100 days, and epsilon is $\frac{1}{2}$. What is the maximum value that the total loss incurred by the algorithm can be?

(b) What value of ϵ should we choose to minimize the total regret, given that we run the algorithm for 25 days?

(c) We run the randomized multiplicative weights algorithm with two experts. In all of the first 140 days, Expert 1 has cost 0 and Expert 2 has cost 1. If we chose $\epsilon = 0.01$, on the 141st day with what probability will we play Expert 1? (Hint: You can assume that $0.99^{70} = \frac{1}{2}$)

Solution:

(a) total regret = loss of algorithm - offline optimum $\leq \epsilon T + \frac{\ln(n)}{\epsilon}$.

$$\text{loss of algorithm} - 20 \leq \frac{1}{2}(100) + \frac{\ln(2)}{\frac{1}{2}}$$

$$\text{loss of algorithm} \leq 50 + 2\ln(2) + 20$$

The maximum loss is roughly 71.39.

(b)

$$R_T \leq \epsilon T + \frac{\ln(n)}{\epsilon}$$

$$R_T \leq \epsilon 25 + \frac{\ln(2)}{\epsilon}$$

Take the derivative with respect to epsilon and set the derivative to 0 to get:

$$25 - \frac{\ln(2)}{\epsilon^2} = 0$$

$$25 = \frac{\ln(2)}{\epsilon^2}$$

$$\epsilon = \sqrt{\frac{\ln(2)}{25}}$$

ϵ should be roughly 0.17.

(c) The weight assigned to expert 1 is $.99^{0 \cdot 140} = 1$, while the weight assigned to expert 2 is $.99^{1 \cdot 140} \approx 1/4$. So, the probability we play expert 1 is $\frac{1}{1+1/4} = 4/5$.