CS 170 HW 14 (optional)

Due **2021-12-14, 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

2 True/False + Justification

For the following parts, state clearly whether the given statement is true or false and give a brief justification for your answer.

1. The node with the highest pre-order number in a directed graph is in a sink SCC.
2. Write a dynamic programming recurrence to find the total number of different ways we can multiply 4 matrices. Recall that matrix multiplication is not commutative, but it is associative, i.e. $(A(B(CD))) = ((A(BC))D)$.
3. If there exists a problem in NP that is neither NP-complete nor in P, $P \neq NP$.
4. $\mathcal{H} = \{x + a \bmod 2n : \forall a \in [2n]\}$ is a universal hash family.
5. Morriss counting algorithm roughly stores the value $x/10$ where x is the number of elements in the stream.

Solution:

1. False. Consider the graph $A \to B \leftarrow C$, and run DFS from A. Then, C has the lowest pre-order number but is not a sink SCC. If we want to find a sink SCC, we can run DFS on the reversed graph and find the node with highest post-order number in the reversed graph.
2. $f(i) = \begin{cases} f(i+2) + f(i+1) & i < 3 \\ 1 & i = 3 \end{cases}$
   The subproblem represents the number of ways we can multiply $4 - i + 1$ matrices.
3. True, if such a problem exists, then $P \neq NP$, since that means P is a strict subset of NP.
4. False. Let $y = 2n + x$. Then, $Pr[h(x) = h(y)] = 1$ for any $h \in \mathcal{H}$.
5. False, Morriss counting algorithm approximately stores the value $\ln(x + 1)$.

3 The Greatest Roads in America

Arguably, one of the best things to do in America is to take a great American road trip. And in America there are some amazing roads to drive on (think Pacific Crest Highway, Route 66 etc). An intrepid traveler has chosen to set course across America in search of some amazing driving. What is the length of the shortest path that hits at least k of these amazing roads?

Assume that the roads in America can be expressed as a directed weighted graph $G = (V, E, d)$, and that our traveler wishes to drive across at least k roads from the subset $R \subseteq E$ of "amazing" roads. Furthermore, assume that the traveler starts and ends at her home $h \in V$. You may also assume that the traveler is fine with repeating roads from R, i.e. the k roads chosen from R need not be unique.

Design an efficient algorithm to solve this problem. Provide a 3-part solution with runtime in terms of $n = |V|$, $m = |E|$, k.

Hint: Create a new graph G' based on G such that for some s', t' in G', each path from s' to t' in G' corresponds to a path of the same length from h to itself in G containing at least k roads in R. It may be easier to start by trying to solve the problem for $k = 1$.

Solution: Main idea:

We want to build a new graph G' such that we can apply Dijkstra's algorithm on G' to solve the problem.

We'll start by creating $k+1$ copies of G. Call these $G_0, G_1, \ldots G_k$. These copies include all the edges and vertices in G, as well as the same weights on edges. Let the copy of v in G_i be denoted by v_i. For each road $(u, v) \in R$, we also add the edges $(u_0, v_1), (u_1, v_2), \ldots (u_{k-1}, v_k)$, with the same weight as (u, v). The intuition behind creating these copies is that each time we use an edge in G' corresponding to an edge in R, we can advance from one copy of G to the next, and this is the only way to advance to the next copy. So if we've reached v_i from h_0, we know we must have used (at least) i edges in R so far.

Now, consider any path p' from h_0 to h_k in G'. If we take each edge (u_i, v_i) or (u_i, v_{i+1}) and replace it with the corresponding edge (u, v) in G, we get a path p in G from h to itself. Furthermore, since the path goes from h_0 to h_k, it contains k edges of the form (u_i, v_{i+1}), where (u, v) is an edge in R. So, p will contain at least k edges in R.

Our algorithm is now just to create G' as described above, and find the shortest path p' from h_0 to h_k using Dijkstra's, and then output the corresponding path p in G.

Correctness:

Assume there is a valid path p in G that is shorter than the one produced by this algorithm. Consider the equivalent path p' in G' formed by modifying the path to go to the next copy of G whenever an edge of R is crossed. Since p is valid, p' must go from h in G_0 to h in G_k. But then p' would be a shorter path in G' than the one produced by Dijkstra's, which is a contradiction.

Runtime:

Since G' includes $k+1$ copies of G, Dijkstra's algorithm will run in time $O((km+kn)\log(kn))$. Since $k \leq m$ and $\log m = O(\log n)$, the runtime can be simplified to $O(k(m+n)\log n)$.

4 Triangulating a polygon

You are given a convex polygon, P, of n vertices, $(x_1, y_1), \ldots, (x_n, y_n)$. A triangulation of P is a collection of $n-3$ diagonals of P such that no two diagonals intersect inside the polygon. A triangulation splits the polygon's interior into $n-2$ disjoint triangles. The cost of a triangulation is defined to be the sum of the lengths of the diagonals forming the triangulation. Your task is to devise a dynamic programming algorithm to compute the minimum cost triangulation of a given polygon. Please provide a three part solution describing your algorithm, a proof of correctness and a runtime analysis. (*Hint*: First order the points $(x_1, y_1), \ldots, (x_n, y_n)$ in a clock-wise manner and index each sub-problem with a pair of indices $1 \leq i < j \leq n$).

Solution: We will denote the points as $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ and assume that the points are arranged in a clock-wise manner. Furthermore, we will denote the distance between p_i and p_j by $d(p_i, p_j)$.

Main Idea:

We will index our sub-problems, $A(i, j)$, with pairs of indices i, j such that $1 \leq i < j \leq n$. The subproblem, $A(i, j)$, will denote the minimum cost triangulation of the polygon formed by the vertices $(x_i, y_i) \ldots (x_j, y_j)$. Our recursive call will be as follows:

$$A(i,j) = \begin{cases} \min(\min_{k \in \{i+2,\ldots,j-2\}}(A(i,k) + A(k,j) + d(p_i,p_k) + d(p_k,p_j)), \\ \qquad d(p_j,p_{i+1}) + A(i+1,j), d(p_i,p_{j-1}) + A(i,j-1)), & \text{if } j-i > 2 \\ 0, & \text{otherwise} \end{cases}$$

At the end of the algorithm, we will output the value $A(1, n)$.

Runtime Analysis:

Notice, that the cost to compute each sub-problem is $O(n)$ and each sub-problem needs to be computed exactly once. This means we can bound the total runtime by $O(n^3)$ as there are $\binom{n}{2}$ choices for i and j.

Proof of Correctness:

We will prove the correctness of our algorithm by induction on the value of $j - i$.

Base Case: $j - i \leq 2$ In the base case, we have three or fewer vertices in our polygon. This implies our polygon is already triangulated.

Inductive Step: Suppose now that we wish to find a triangulation of the polygon, P_{ij}, formed by the vertices $p_i, \ldots, p_j$. Now consider the edge (j, i). In any triangulation of P_{ij}, the edge (j, i) has to be part of some triangle. The third vertex of the triangle is one of the vertices, $p_{i+1}, \ldots, p_{j-1}$. The cases where the third vertex of the triangle is one of the vertices p_{i+1} or p_{j-1} is covered by the last two terms in the recursive formula as in both cases we add either the edge $(j, i+1)$ or $(i, j-1)$. For all the other possibilities (where $k \in \{i+2, \ldots, j-2\}$), we add both edges (i, k) and (j, k) which is covered by the first term of the recursion. In particular, for the optimal triangulation of P_{ij}, the edge (j, i) is part of some triangle, say i, j, k^*. We see that this triangulation consists of a triangulation of P_{ik^*} and P_{k^*j} along with the edges (i, k^*) and (k^*, j). We see that the cost computed by our algorithm is at most the cost of this triangulation. The correctness of our algorithm is evident from the fact that we may construct a triangulation from the choice of k minimizing the recursive formula.

5 Steel Beams

You're a construction engineer tasked with building a new transit center for a large city. The design for the center calls for a T-foot-long steel beam for integer $T > 0$. Your supplier can provide you with an *unlimited* number of steel beams of integer lengths $0 < c_1 < \ldots < c_k$ feet. You can weld as many beams as you like together; if you weld together an a-foot beam and a b-foot beam you'll have an $(a + b)$-foot beam. Unfortunately, every weld increases the chance that the beam might break, so you want as few as possible.

Your task is to design an algorithm which outputs how many beams of each length you need to obtain a T-foot beam with the minimum number of welds, or 'not possible' if there's no way to make a T-foot beam from the lengths you're given. (If there are multiple optimal solutions, your algorithm may return any of them.)

(a) Consider the following greedy strategy. Start with zero beams of each type. While the total length of all the beams you have is less than T, add the longest beam you can without the total length going over T.

  (i) Suppose that we have 1-foot, 2-foot and 5-foot beams. Show that the greedy strategy always finds the optimum.

  (ii) Find a (short) list of beam sizes $c_1, \ldots, c_k$ and target T such that the greedy strategy fails to find the optimum. Briefly justify your choice.

(b) Give a dynamic programming algorithm which always finds the optimum.

  (i) State your recurrence relation.

  (ii) Prove correctness of your algorithm by induction.

    i. Show that the base case is correct.

    ii. Assuming that your recurrence relation is correct for previous subproblems, show that it gives the correct value for the current subproblem.

    iii. Give the order where you can solve the subproblems, and show that for this order, evaluating the recurrence relation will use only subproblems that have already been computed.

  (iii) Find the running time and space requirement of your algorithm.

Solution: Formal statement of problem: Given a list of integers $C = (c_1, \ldots, c_k)$ with $0 < c_1 < \ldots < c_k$ and a target $T > 0$, the algorithm should output **nonnegative** integers $(a_1, \ldots, a_k)$ such that $\sum_{i=1}^{k} a_i c_i = T$ where $\sum_{i=1}^{k} a_i$ is as small as possible, or return 'not possible' if no such integers exist.

(a) (i) Let a_1, a_2, a_3 be some optimum solution. We know that $a_1 < 2$ since if $a_1 \geq 2$ we can improve the solution by taking $a_1 - 2, a_2 + 1, a_3$. If $a_1 = 1$ then $a_2 < 2$ because otherwise we could improve by taking $0, a_2 - 2, a_3 + 1$. So the possible values of the optimum are $0, 1, j$, $0, 2, j$, $1, 1, j$ for some $j \geq 0$. In each case the greedy algorithm would give the same answer.

  (ii) $C = (4, 5)$, $T = 8$ is one possibility.

(b) (i) We create a dynamic programming algorithm where, for each $n \leq A$, we will find the minimum integer combination that sums to n. The recurrence is $f(n) = \min_{i=1}^{k} f(n - c_i) + 1$, with $f(0) = 0$.

(ii) i. The base case is $f(0) = 0$.

ii. Fix $n > 0$. Suppose that $f(n')$ is optimal for all $n' < n$. Firstly note that if $a'_1, \ldots, a'_k$ is a minimum integer combination summing to $n - c_i$, then $a'_1, \ldots, a'_i + 1, \ldots, a'_k$ is an integer combination summing to n (not necessarily minimum). Let $a_1, \ldots, a_k$ be a minimum integer combination summing to n. Then for every i with $a_i > 0$, $a_1, \ldots, a_i - 1, \ldots, a_k$ is a minimum integer combination summing to $n - c_i$ (otherwise $a_1, \ldots, a_k$ wouldn't be optimal). We know that some $a_i > 0$ (we don't know which), so we take the minimum over all i.

iii. We solve the subproblems in the order of smallest to largest n, since to compute $f(n)$, we only use $f(n')$ for $n' < n$.

(iii) We compute T subproblems, each one being a minimum of k values, so the running time is $O(Tk)$. The space requirement is $O(T)$.

In addition, we can build the optimal solution in extra $O(T)$ time and $O(T)$ space as follows: Each subproblem will also return the type of beam added, and a pointer to the previous subproblem it is added to. The solution can be built by traversing the chain of pointers from $f(T)$ to the base case in $O(T)$ time.

6 Survivable Network Design

Survivable Network Design is the following problem: We are given two $n \times n$ matrices: a cost matrix d_{ij} and a (symmetric) connectivity requirement matrix r_{ij}. We are also given a budget b. We want to find a undirected graph $G = (\{1, ..., n\}, E)$ such that the total cost of all edges (i.e. $\sum_{(i,j)\in E} d_{ij}$) is at most b and there are exactly r_{ij} edge-disjoint paths between any two distinct vertices i and j, or if no such G exists, output "None". (A set of paths is edge-disjoint if no edge appears in more than one of them)

Show that Survivable Network Design is NP-Complete. (Hint: Reduce from a NP-Hard problem in Section 8 of the textbook.)

Solution: To show Survivable Network Design is NP. Given a solution to Survivable Network Design (i.e. a graph G), we can check the budget constraint in linear time easily. To check the connectivity requirements, we can check if the max flow between every pair i, j using unit capacities on the edges is at least r_{ij}. Any max-flow of value F can be decomposed into F edge-disjoint i-j paths, since all capacities are unit.

We reduce from Rudrata Cycle to Survivable Network Design: Given $G = (V, E)$, take $b = n$ (recall $|V| = n$), $d_{ij} = 1$ $\forall (i, j) \in E$ and $d_{ij} > n$ otherwise; set $r_{ij} = 2$ $\forall (i, j)$.

Clearly any Rudrata Cycle is a solution to this Survivable Network Design instance. So we just need to show any solution in this instance is a Rudrata Cycle.

Any solution uses edges such that the sum of the weights of all the edges is at most n, i.e. uses at most n edges since all usable edges have weight 1. In order to have two edge-disjoint paths between any two distinct vertices, every vertex must have degree at least 2. But since we can only use n edges, the total degree of all vertices can't be more than $2n$. So every

vertex has degree exactly 2. This means that the edges must form a set of disjoint cycles. But every pair of vertices needs to be connected by the edges, so this set of disjoint cycles must actually be a single cycle visiting every vertex, i.e. a Rudrata Cycle.

7 Coffee Shops

A rectangular city is divided into a grid of $m \times n$ blocks. You would like to set up coffee shops so that for every block in the city, either there is a coffee shop within the block or there is one in a neighboring block. (There are up to 4 neighboring blocks for every block). It costs r_{ij} to rent space for a coffee shop in block ij.

Write an integer linear program to determine which blocks to set up the coffee shops at, so as to minimize the total rental costs.

(a) What are your variables, and what do they mean?

(b) What is the objective function?

(c) What are the constraints?

(d) Solving the non-integer version of the linear program gets you a real-valued solution. How would you round the LP solution to obtain an integer solution to the problem? Describe the algorithm in at most two sentences.

(e) What is the approximation ratio obtained by your algorithm?

(f) Briefly justify the approximation ratio.

Solution:

(a) There is a variable for every block x_{ij}, i.e., $\{x_{ij} | i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}\}$. This variable corresponds to whether we put a coffee shop at that block or not.

(b) $\min \sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij} x_{ij}$. Alternatively, $\min t$ is correct as well as long as the correct constraint is added.

(c) (i) $x_{ij} \geq 0$: This constraint just corresponds to saying that there either is or isn't a coffee shop at any block. $x_{ij} \in \{0, 1\}$ or $x_{ij} \in \mathbb{Z}_+$ is also correct.

(ii) For every $1 \leq i \leq m, 1 \leq j \leq n$:

$$x_{ij} + x_{(i+1),j} \mathbb{1}_{\{i+1 \leq m\}} + x_{(i-1),j} \mathbb{1}_{\{i-1 \geq 1\}} + x_{i,(j+1)} \mathbb{1}_{\{j+1 \leq n\}} + x_{i,(j-1)} \mathbb{1}_{\{j-1 \geq 1\}} \geq 1$$

This constraint corresponds to that for every block, there needs to be a coffee shop at that block or a neighboring block.

$\mathbb{1}_{\{i+1 \leq m\}}$ means "1 if $\{i+1 \leq m\}$, and 0 otherwise". It keeps track of the fact that we may not have all 4 neighbors on the edges, for instance.

(iii) If the objective was $\min t$, then the constraint $\sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij} x_{ij} \leq t$ needs to be added.

(d) Round to 1 all variables which are greater than or equal to 1/5. Otherwise, round to 0.

In other words, put a coffee shop on (i, j) iff $x_{i,j} \geq 0.2$.

(e) Using the rounding scheme in the previous part gives a 5-approximation.

(f) Notice that every constraint has at most 5 variables. So for every constraint, there exists at least one variable in the constraint which has value $\geq 1/5$ (not everyone is below average). The total cost of the rounded solution is at most $5 \cdot$ LP-OPT, since $r_{ij} \leq 5 r_{ij} x_{ij}$ for any x_{ij} that gets rounded up, and the other i, j pairs contribute nothing to the cost of the rounded solution. Since Integral-OPT $\geq$ LP-OPT (the LP is more general than the ILP), our rounding gives value at most 5 LP-OPT $\leq$ 5 Integral-OPT. So we get a 5-approximation.

8 Porcupine Trio

Let L be a vector of integers in $[-M, M]^n$ given to us as input. For any other vector x, we will use $\langle L, x \rangle$ to denote $\sum_{i=1}^{n} L_i x_i$. In this problem we will assume $M < 2^n/(4n)$.

(a) Prove that there exists distinct $x_1, x_2 \in \{\pm 1\}^n$ such that $\langle L, x_1\rangle = \langle L, x_2\rangle$.
Hint: try to prove this using the pigeonhole principle. **Solution:** $\langle L, x\rangle$ is always in $[-Mn, Mn]$ and hence with our bound on M there are strictly less than 2^n possibilities for $\langle L, x\rangle$. Since there are exactly 2^n possibilities for x, by the pigeonhole principle there must exist distinct x_1, x_2 such that $\langle L, x_1\rangle = \langle L, x_2\rangle$.

(b) Let $\boldsymbol{x}$ be sampled uniformly at random from $\{\pm 1\}^n$. Use Chebyshev's inequality to prove that:

$$\Pr[|\langle L, \boldsymbol{x}\rangle| > 10M\sqrt{n}] \leq \frac{1}{100}.$$

Solution: $\mathbf{E}[\langle L, \boldsymbol{x}\rangle^2] = \sum_{i=1}^n L_i^2 \leq M^2 n$. Since $\mathbf{E}[\langle L, \boldsymbol{x}\rangle] = 0$, we know $\sqrt{\mathbf{Var}[\langle L, \boldsymbol{x}\rangle]} \leq M\sqrt{n}$. The desired inequality is then a direct consequence of this chatter combined with Chebyshev's inequality.

(c) Let $\boldsymbol{Y}$ be a random variable that is equal to 1 with probability $\frac{1}{k}$ and 0 otherwise. Let $Y_1, \ldots, Y_{2k\ln r}$ be $2k \ln r$ independent copies of $\boldsymbol{Y}$. Prove:

$$\Pr[\boldsymbol{Y}_1 + \cdots + \boldsymbol{Y}_{2k\ln r} < 2] \leq \frac{2}{r}.$$

Hint: you may use the fact that $\left(1 - \frac{1}{k}\right)^k \leq \frac{1}{e}$ *without proof.* **Solution:** $\Pr[\boldsymbol{Y}_1 + \cdots + \boldsymbol{Y}_{k\ln r} = 0] = \left(1 - \frac{1}{k}\right)^{k\ln r} \leq \frac{1}{r}$. Identically, $\Pr[\boldsymbol{Y}_{k\ln r+1} + \cdots + \boldsymbol{Y}_{2k\ln r} = 0] = \left(1 - \frac{1}{k}\right)^{k\ln r} \leq \frac{1}{r}$. A union bound on the two tells us that the probability that "at least one of the sums is 0 is at most $\frac{2}{r}$". The event in the quotations "" is a superset of the event whose probability we wish to bound as if the total sum is ¡ 2 at least one of the halves must sum to 0, which complete the proof.

(d) Give a randomized algorithm that runs in time $O(Mn^{3/2}\log(n))$ and with probability $1 - o_n(1)$ outputs distinct $x_1, x_2 \in \{\pm 1\}^n$ such that $\langle L, x_1\rangle = \langle L, x_2\rangle$. **Solution:** For $\boldsymbol{x} \sim \{\pm 1\}^n$, the most likely outcome P for $\langle L, \boldsymbol{x}\rangle$ in $[-10M\sqrt{n}, 10M\sqrt{n}]$ has probability at least $\frac{1}{\ell} := \frac{.99}{20M\sqrt{n}+1}$ from part b. If we sample $\lceil 2\ell \ln n\rceil$ independent $\boldsymbol{x}$, by part c at least two of the samples $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ satisfy $\langle L, \boldsymbol{x}_1\rangle = \langle L, \boldsymbol{x}_2\rangle = P$ with probability at least $1 - \frac{1}{n}$. Now conditioned on this $1 - \frac{1}{n}$ probability event, we turn our attention to lower bounding the probability that $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ (the first two samples satisfying $\langle L, \boldsymbol{x}_1\rangle = \langle L, \boldsymbol{x}_2\rangle = P$ are *distinct*. The number of different x such that $\langle L, x\rangle = P$ is at least $\frac{2^n}{\ell} \geq C\sqrt{n}$ for some constant C. Thus, the probability that $\boldsymbol{x}_2 \neq \boldsymbol{x}_1$ is at least $1 - \frac{1}{C\sqrt{n}}$. To summarize, with probability at least $\left(1 - \frac{1}{n}\right) \cdot \left(1 - \frac{C}{\sqrt{n}}\right)$ there will be two distinct samples $\boldsymbol{x}_1, \boldsymbol{x}_2$ among $\lceil 2\ell \ln n\rceil$ independent samplies such that $\langle L, \boldsymbol{x}_1\rangle = \langle L, \boldsymbol{x}_2\rangle = P$.

This suggests the following algorithm: sample $\lceil 2\ell \ln n\rceil$ independent $\boldsymbol{x} \sim \{\pm 1\}^n$, compute each $\langle L, \boldsymbol{x}\rangle$, store pairs $(\boldsymbol{x}, \langle L, \boldsymbol{x}\rangle)$ in a list, and then sort the list by $\langle L, \boldsymbol{x}\rangle$ values. Iterate through the list, and output the first adjacent pair $\boldsymbol{x}_i, \boldsymbol{x}_{i+1}$ such that $\langle L, \boldsymbol{x}_i\rangle = \langle L, \boldsymbol{x}_{i+1}\rangle$ and $\boldsymbol{x}_i \neq \boldsymbol{x}_{i+1}$.

(e) (Fun bonus question worth no points) Can you improve the runtime in the previous part to $O(\sqrt{M}n^{5/4}\log n)$?