This HW has four regular problems. All problems on the HWs are to be done individually, no collaboration is allowed. Solutions to the written problems should be submitted as a single PDF file using gradescope, with the answer to each problem starting on a new page.

There is no programming problem this HW: we’ll be back next HW!

However there is a bonus problem. You won’t get hints or assistance from the course staff, but you can work together in groups. (Each group should submit only once and include the names of all the collaborators on the submission.) It is worth 1% under the quiz/bonus category if you solve it correctly. No partial credit.

(25 pts.) 1. **Load Balancing with Picky Jobs.** You are given \(n\) jobs (each job \(j\) having size \(p_j\)) and \(m\) machines, and you want to assign jobs to machines to minimize the makespan (the max load of any machine). But here’s the catch: not every job can be assigned to every machine. (E.g., some machines don’t have the power to work on some jobs, etc.) In fact, there is a bipartite graph \((M, J, E)\), where \(|J| = n\) are the jobs, \(|M| = m\) are the machines, and there is an edge \((i, j) \in E\) if job \(j\) can be assigned to machine \(i\).

Let \(OPT\) be the makespan of the optimal schedule.

(a) Show that for every pair of integers \(m, n\) with \(m \geq n\), there exists an instance where the greedy algorithm (which just considers the jobs in the given order \(p_1, p_2, \ldots, p_n\)) and assigns them to an arbitrary least loaded machine thus far) can result in a load of \(\log_2 n \cdot OPT\).

(b) To do this assignment, we write the following LP:

\[
\begin{align*}
\text{min} & \quad L \\
\text{such that} & \quad \sum_{j \in J : (i, j) \in E} p_j x_{ij} \leq L & \forall i \in M \\
& \quad \sum_{i \in M : (i, j) \in E} x_{ij} = 1 & \forall j \in J \\
& \quad x_{ij} \geq 0
\end{align*}
\]

Let \((x^*, L^*)\) be an optimal (vertex) solution to this LP. Show that \(L^* \leq OPT\).

(c) The support of \(x^*\) is the set of edges \((i, j) \in E\) on which \(x_{ij}^* \neq 0\). If \(x^*\) is a vertex solution, show that its support cannot have any cycles. (Hint: recall the argument in Lecture #14, Section 7.1.)

(d) Since the support of \(x^*\) is a forest \(F \subseteq E\), for each tree \(T \in F\) do the following: root \(T\) at some vertex corresponding to a job. Now assign each job \(j\) in \(T\) to an arbitrary one of the machines which are its children in \(T\). If \(j\) has no child, then assign it to its parent machine.
Show that the total load of any machine $i$ after this process is at most

$$L^* + \max_{j: x_{ij} \neq 0} p_j \leq 2OPT.$$ 

**Solution:**

(a) Let’s consider the case $|J| = |M| = n$ (if $m > n$ just let the rest machines being idle). We organize the jobs into groups. Group $i$ contains $\lceil n/2^i \rceil$ jobs: the first $\lceil n/2 \rceil$ jobs are in group 1; the next $\lceil n/4 \rceil$ jobs are in group 2, etc. We then construct a graph with edges:

i. Connect $(j, m_{n-i})$

ii. Connect the $i$-th job in any group $g$ to machine $i$.

An example with $n = 7$ is shown in Figure 1 (a). An observation is that the jobs in group $i$ are always connected to the first $n/2^i - 1$ machines.

The optimal solution, obviously, is to assigned job $i$ to machine $n - i$, as shown in Figure 1 (c). The optimal load is 1. We inductively show that it is possible that after all jobs in $i$-th group are assigned, the first $\lceil n/2^i \rceil$ machines can have work load $i$.

For the first group, a job $j$ can be assigned to two machines: one in the first half (machine $j$), and one in the second half (machine $n - j$). At the beginning each machine has load 0. The algorithm can assigned each job $j$ to machine $j$. After that, the first $n/2$ machines have load 1.

Assume it holds for group $i - 1$, we now show it is still true for $i$. The $j$-th job in group $i$, which among all machines should have a rank of $\lceil n/2 \rceil + \lceil n/4 \rceil + \ldots + \lceil n/2^i \rceil + j = n(1 - 1/2^i) + j$, has two choices: machine $j$, and machine $\lceil 1/2^i \rceil - j$. Both are in the first $\lceil n/2^{i-1} \rceil$ machines. From the hypothesis they have the same load $i - 1$, and it is possible that job $j$ is assigned to machine $j$, which makes the first $\lceil n/2^i \rceil$ machines to have load $i$.

Thus, after all $\log n$ groups, the first machine will have load $\log n$. An example for $n = 7$ is shown in Figure 1 (b).

Then we show the work load achieved by the greedy assignment can be as bad as $\log n$.

(b) The $OPT$ is one feasible solution to the LP. Hence $L^* \leq OPT$.

(c) Assume to the contrary that there is a cycle in the support. First we show that all the edges on the cycle have weight strictly less than 1. Any edge with weight 1 assigning a job $j$ to a machine $m$ must be the only edge connected to $j$ in the support because of the constraint $\sum_{i \in M: (i,j) \in E} x_{ij} = 1$. This means that this edge cannot be an edge in a cycle.

Now we know that in a cycle, all edges have weight in range $(0, 1)$. Then we can adopt the proof of Theorem 1 in Lecture 14 to show that it is not a vertex solution. However we have another variable $L$ here so the proof is slightly different.

For any cycle in this bipartite graph, we know that it must be an even cycle. On this cycle the nodes are alternately jobs and machines. Suppose the length of the cycle $C$ is $2k$. WLOG, assume the nodes on the cycle are $m_1, j_1, m_2, j_2, \ldots, m_k, j_k$.

The variables related to this cycle are $x_{11}, x_{21}, x_{22}, x_{32}, \ldots, x_{kk}, x_{1k}$. We use a similar
The bipartite graph

(b) Greedy solution = 3 = \(\lceil \log_2 n \rceil\)

(c) OPT = 1

Group 1

Group 2

Group 3

Figure 1: Illustration for Problem 1 (a).

method in Lecture 14, which assigns \(\epsilon/p_j\) or \(-\epsilon/p_j\) to variable \(x_{ij}\). All the other variables stay unchanged. We denote the new variables as \(x'_{ij}\).

The second constraint \(\sum_{i \in M,(i,j) \in E} x_{ij} = 1\) holds because for each job \(j\) on the cycle, only two variables \(x_{jj}\) and \(x_{j+1,j}\) are affected, and the new value \(x'_{jj} + x'_{j+1,j} = (x_{jj} \pm \epsilon/p_j) + (x_{j+1,j} \mp \epsilon/p_j) = x_{jj} + x_{j+1,j}\), which is unchanged.

The First constraint still holds because for each machine, only two variables \(x_{i,(i-1)}\) and \(x_{ii}\) are affected, and the new value \(p_{i-1}x'_{i,(i-1)} + p_{i}x'_{ii} = p_{i-1}(x_{i,(i-1)} \pm \epsilon/p_{i-1}) + p_{i}(x_{ii} \mp \epsilon/p_{i}) = p_{i-1}x_{i,(i-1)} + p_{i}x_{ii}\).

This means that the new \(x_{ij}\) with \(\epsilon\) added or subtracted and the old \(L\) is still feasible.

Then, similar as in the lecture notes, we find a vector \(v = \langle \epsilon/p_1, -\epsilon/p_1, \epsilon/p_2, -\epsilon/p_2, \ldots, 0, \ldots, 0 \rangle\) such that the point \(q + v\) and the point \(q - v\) are both feasible (\(q\) is the solution with cycles in its support).

(d) As defined in the problem, every non-leaf job will be assigned to an arbitrary child, and leaves will be assigned to their parent. In this case, if a machine is not the parent of a leaf node, it will only take 1 job, which is its parent. For those machines, the work load \(W \leq \max_{j \in S_m} x^*_{ij} p_i \leq L^* + \max_{j \in S_m} x^*_{ij} p_j\).

For a machine \(m\) that has more than one jobs, it must be the parent of a leaf. All the jobs it has are the jobs of all its children and also, its parent. Let’s first look at its children. We denote the set of children of machine \(m\) as \(S_m\). In the support of \(x^*\), these jobs have no other neighbors except \(m\) (otherwise they are not leaves). Thus in \(x^*\) the weight of edges \((j', m)\) where \(j' \in S_m\) is exactly 1. Thus in the optimal solution \(L^*\) is at least \(\sum_{j \in S_m} p_j\).

Then this machine has another job to do, which is its parent. The size of the job is no more than \(\max_{j \in S_m} x^*_{ij} p_j\).

The work load of the machine \(m\) with more than one jobs is \(W = \sum_{j \in S_m} p_j + p_i \leq L^* + \max_{j \in S_m} x^*_{ij} \neq 0 p_j\). (\(i\) is \(m\)'s parent).

We have shown in (b) that \(L^* \leq \text{OPT}\). Also, assume \(j^* = \arg \max_{j \in S_m} x^*_{ij} \neq 0 p_j\). Then this job \(j\) must be assigned to some machine in the \(\text{OPT}\). Thus \(p_j \leq \text{OPT}\). So \(L^* + \max_{j \in S_m} x^*_{ij} \neq 0 p_j \leq 2\text{OPT}\).
Sudoku is so Old-School. In Generalized Sudoku we are given \( n \) items, a positive integer \( k \), and \( m \) subsets \( S_1, S_2, \ldots, S_m \) of these items such that \( |S_i| = k \) for all \( i \). The goal is to give each item a label from \( \{1, \ldots, k\} \) so that every subset \( S_i \) contains exactly one item of each label. (So, in standard Sudoku, the items are arranged in a \( 9 \times 9 \) grid, the sets \( S_i \) are the rows, columns and \( 3 \times 3 \) mini-grids, and \( k = 9 \).) Specifically, let’s define the decision version of Generalized Sudoku to be: given the sets \( S_1, S_2, \ldots, S_m \) and the integer \( k \), does a solution of the desired form exist? The search version of the problem is to actually find a solution of the desired form, if one exists.

(a) Prove that the decision version of Generalized Sudoku is in NP.

(b) Prove that the decision version of Generalized Sudoku is NP-hard for \( k \geq 3 \). (This, combined with (a), means it is NP-Complete.)

(c) Prove that Generalized Sudoku is in P for the case \( k = 2 \). In particular, give a polynomial-time algorithm to determine if a solution of the desired form exists, and to find such as solution if one indeed does exist (i.e., solve the search version).

Solution: (a) First, we reduce the 3-coloring problem to Sudoku with \( k = 3 \). Specifically, given a graph \( G = (V, E) \), let the items be \( V \cup E \), and for each \( e = (u, v) \in E \) let \( S_e = \{u, v, e\} \) (i.e. for each edge, create a set containing the edge and its two endpoints). If \( G \) has a 3-coloring, say using colors \( \{1, 2, 3\} \), then by giving each edge the label that is not used by its endpoints we have a legal solution to the Sudoku problem. In the other direction, given a solution to the Sudoku problem, the labels used on the vertices must be a legal 3-coloring of \( G \) since no edge will have both its endpoints the same color. Since we know that 3-coloring is NP-complete, this implies that Generalized Sudoku with \( k = 3 \) is NP-hard.

Now, we can inductively show Generalized Sudoku is NP-hard for any \( k \geq 3 \). Suppose we know Generalized Sodoku is NP-complete for \( k-1 \). Then we can reduce this to Generalized Sodoku for \( k \) by adding an extra element, \( k \), to every set. Clearly, we can label the new sets with \( k \) labels iff the old sets could be labeled with \( k - 1 \).

(b) Given an instance of Generalized Sudoku with \( k = 2 \), we can create a graph \( G \) with one vertex for each item and an edge \((u, v)\) for each corresponding set \( \{u, v\} \). This graph will be bipartite if and only if the given instance of Generalized Sudoku has a solution. Specifically, if the given instance has a solution, then putting all nodes labeled 1 on the left and all nodes labeled 2 on the right is a legal bipartition; in the reverse direction, given a bipartition a legal solution is to assign label 1 to all nodes on the left and assign label 2 to all nodes on the right.

We can test if a graph is bipartite in polynomial time: for each component, just do breadth-first search from any node in it and check to see if there are any edges between two nodes at the same level. So, this means the Generalized Sudoku problem with \( k = 2 \) can be solved in polynomial time.

Many Problems? No Problem! You just realize that the end of the semester is looming, and you need to get your act together. Your textbook has \( n \) chapters, each
quite technical, where chapter $i$ costs $C_i$ dollars to read. ("Time is money," as they say.)

The homework consists of a set of $m$ problems $p_1, p_2, \ldots, p_m$. For each problem $p_j$, there is a subset $P_j \subseteq \{1, \ldots, n\}$ of the chapters that it depends on. If you have read all the chapters in $P_j$ you can solve the problem $p_j$, but if you've missed even one of the chapters in $P_j$ you cannot solve the problem. Solving $p_j$ gives you $V_j$ dollars of value. The net utility is the value of the problems solved minus the cost of the chapters read. The goal is to find a subset $R \subseteq \{1, \ldots, n\}$ of the chapters to read, to maximize the net utility you get.

E.g., if $P_1 = \{2, 3, 5\}$, $P_2 = \{1, 2, 3\}$ and $P_3 = \{2, 3, 4\}$. Suppose the costs for the $n = 5$ chapters are 1, 4, 3, 8, 1 respectively, and values for the three problems are $V_1 = 14, V_2 = 4, V_3 = 7$. If you have read chapters $\{2, 3, 4, 5\}$ then your cost is $4 + 3 + 8 + 1 = 16$ and the value you get from having solved $P_1$ and $P_3$ is $14 + 7 = 21$. So the net utility is $21 - 16 = 5$.

On the other hand, having read just 2, 3, 5 you'd have net utility $14 - (4 + 3 + 1) = 6$. And having read just 1, 2 you'd get net utility $0 - (1 + 4) = -5$.

Show how to use an $s$-$t$-min-cut algorithm to solve this problem in polynomial time.

Hint: Can you solve the problem of minimizing the cost of the chapters you read plus the sum of the values of problems you did not solve. Why is solving this problem this useful? Think about how you could solve this problem using an $s$-$t$ min-cut algorithm.

**Solution:** First, the $x$ that maximizes a function $f(x)$ is the same as the $x$ that minimizes the function $-f(x)$, so maximizing the values of problems solved minus the cost of chapters read is the same as minimizing the cost of chapters read minus the values of problems solved. Next, adding a constant offset doesn’t change the location of the maximum, so by adding the sum of values of all problems we get to the equivalent question of finding the chapters to read that minimizes the cost of chapters read plus the values of problems not solved. Let’s call this the penalty of a solution. So, our goal is to find a solution of minimum penalty.

Now, we construct the following flow graph. We have a source $s$ (call this “level 0”), a set of $m$ vertices labeled $p_1, \ldots, p_m$ at level 1, a set of $n$ vertices labeled $ch_1, \ldots, ch_n$ at level 2, and a sink $t$ at level 3. The source is connected to each vertex $p_j$ by an edge of capacity $V_j$. Vertices $p_j$ are connected by an edge of infinite capacity to each vertex $ch_i$ for $i \in P_j$, and each vertex $ch_i$ is connected by an edge of capacity $C_i$ to the sink. All edges are directed left-to-right. We solve for the minimum $s$-$t$ cut, and output as our solution to read all chapters and solve all problems whose associated vertices are on the $s$-side of the cut. (An equally-good construction is to connect the source to the chapters, connect the chapters to the problems, and then connect the problems to the sink, in which case we would output the chapters and problems on the $t$-side of the cut).

To analyze this we need to prove two directions: (1) the solution proposed is legal (we never say to solve a problem whose chapters haven’t all been read) and has penalty equal to the capacity of the cut, and (2) vice-versa: for any proposed solution $Y$, there exists a cut of capacity equal to the penalty of $Y$. Note that we must prove (2) in order to conclude that the minimum cut gives the solution of minimum penalty (otherwise, perhaps there is a better solution that doesn’t correspond to any cut at all).
Proof of (1): the minimum cut will not cut any edge of infinite capacity, which means that if a problem \( p_j \) is on the s-side of the cut, all chapters it depends on will be on the s-side as well. This shows that the solution is legal. Moreover, the capacity of the cut is the capacity of all edges between the source and problems \( p_j \) on the t-side of the cut plus the capacity of all edges from chapters \( ch_i \) on the s-side of the cut to \( t \). This is precisely the penalty of the proposed solution.

Proof of (2): Given a proposed solution \( Y \), we cut edges from \( s \) to problems not solved, and from \( t \) to chapters read. We cut edges of total capacity equal to the penalty of the solution, so what remains is to prove this is an s-t cut. Suppose for contradiction there was a path remaining from \( s \) to \( t \). Since all edges are directed left-to-right, this path must be of the form "\( s-p_j-ch_i-t \)". But this means that we solved problem \( p_j \) without reading the chapter \( ch_i \) that it depended on, contradicting the legality of \( Y \). Note that it is crucial for this argument that the edges in the graph be directed: otherwise there could be a path from \( s \) to a problem solved, to a chapter read, back to a problem not solved, forward to a chapter not read, and then to \( t \).

(25 pts) 4. A Problem with a Problem. The instructors in your course have changed the rules for how they award credit for the course. As before, you have a set of \( m \) problems you can solve, and \( n \) chapters you can read. Problem \( p_j \) can be solved only if you have read all the chapters in set \( P_j \subseteq \{1, 2, \ldots, n\} \). The cost of chapter \( i \) is \( c_i > 0 \). So far the same. But you’re also given an integer \( L > 0 \), and the new rule says: you get a good grade in the course if you solve at least \( L \) problems out of \( n \). As an enterprising student, you’d like to minimize the cost of reading chapters necessary to get a good grade.

The decision problem now is: given \( n \) chapters with costs, \( m \) problems with associated sets of chapters, and parameters \( L \in \{1, 2, \ldots, m\} \) and \( C \in \{1, 2, \ldots, \sum c_i\} \), does there exist a subset of chapters of total cost at most \( C \), so that you can solve at least \( L \) problems having read these chapters.

Prove that this problem is NP-complete.

Solution: First, to show this problem is in NP. Note that if I gave you a set \( S \) of chapters, you could verify that it had cost at most \( C \) (by summing up the costs \( \sum_{i \in S} c_i \) and checking if this sum was at least \( C \)), and that it could be used to solve at least \( L \) problems (just ensure there are at least \( L \) problems which depend only on these chapters in \( S \)).

Now to show NP-hardness. Let us reduce from the Clique problem. Take any instance of the clique problem, which consists of an undirected graph \( G \), and a number \( k \), and asks if there exists a clique in \( G \) of at least \( k \) vertices. From this we want to get an instance of our problem, we will call this instance \( f(G) \). We want that \( G \) is a “Yes” instance of Clique if and only if \( f(G) \) is a “Yes” instance of our problem.

Good. What is this instance \( f(G) \) of our problem? Make one chapter for every node in \( G \), each with cost \( c_i = 1 \). Make one problem for every edge \( (u, v) \in G \). The problem \( (u, v) \) depends on the two chapters \( u \) and \( v \). Set the target \( L \) to be \( \binom{k}{2} \). And the threshold \( C \) to be \( k \). So we are asking: is it possible to read at most \( k \) chapters so that we can solve at least \( \binom{k}{2} \) problems. Note that the maximum number of problems we can solve (by the
way we constructed our instance) by reading \( k \) chapters is \( \binom{k}{2} \), and this is possible exactly when \( G \) has a clique on \( k \) vertices. In other words, \( G \) has a clique on \( k \) vertices if and only if our instance has a “Yes” answer. This completes the NP-hardness proof.

We’ve shown our problem is in NP, and it is NP-hard. Hence it is NP-complete.

(1 pt bonus) 5. **Bonus: Defusing a Dangerous String.** You’re given a string \( S \) of length \( n \) consisting of characters 0, 1, and *. Such a string is considered dangerous if it contains the substring 10, and harmless otherwise. For example *111000 and 010 are dangerous, but 111*000 and 001 are harmless.

You’re allowed to do swaps of neighboring elements to render a given dangerous string harmless. For example the string *111000 can be rendered harmless by doing three swaps, but the string 111000 requires nine swaps.

Give a polynomial-time algorithm to compute the minimum number of swaps necessary to do this.

(1 pt bonus) 6. **Bonus: Covering a Matrix.** You’re given an \( n \times m \) matrix of 0s and 1s. The problem is to pick some columns and rows so that the union of these covers all the ones in the matrix. For example, consider the following matrix:

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{bmatrix}
\]

Rows 2, 3, 4, and 5 suffice to cover all the 1’s. Alternatively rows 2 and 4, along with columns 4 and 5 suffice. (There are other combinations as well.) We consider two specific criteria for judging the quality of a solution.

(a) Suppose the goal is to minimize the number of rows plus number of columns chosen. Show how to solve this in polynomial time.

(b) Suppose the goal is to minimize the maximum of the number of rows and columns chosen. (In the above example the answer would be 2.) Prove that the decision version of this problem is NP-complete. That is, given a \( n \times m \) boolean matrix and a number \( k \), is it possible to choose at most \( k \) rows and at most \( k \) columns so that all the 1s are covered.