

- Submit on Gradescope within **24 hours of your first access**. Please note down the time of your access now on the top of this page, above the Final heading.
- Accesses are logged by the system. **Your timeliness is expected**.
- Timestamps in the form of carefully, completely-photographed work sent by email are admissible in a technical emergency.
- Do not share or make the exam accessible to anyone other than yourself. Any suspected violation of this policy will constitute a breach of the University's academic integrity policy and will be reported.
- You can either type your solutions using LaTeX or scan your handwritten work. You will find a LaTeX template in the final folder. If you are writing by hand, please fill in the solutions in this template, inserting additional sheets as necessary. This will facilitate the grading.
- **No collaboration and no use of external resources** are allowed for this final. Any suspected violation will be investigated and reported. You may consult the textbook, the material and videos linked from the Canvas Modules page and any notes that you took for the course.
- If you absolutely need to contact the teaching team, please use a private message on Ed.

PROBLEM 1 (Reduction Between Flows). Given a directed graph $G = (V, E)$, consider finding the maximum flow in the following variant scenario

- (a) In addition to edge capacities c_e in your flow network, every vertex $v \in V$ has a vertex capacity d_v , and the flow $f : E \rightarrow \mathbb{R}_{\geq 0}$ must satisfy for all $v \in V$

$$\sum_{u:(u,v) \in E} f_{u,v} \leq c_v \quad (1)$$

Show that this problem reduces to the maximum flow problem as discussed in class. Precisely, given a directed graph $G = (V, E)$ satisfying the variant constraint above, give a construction of another directed graph $G' = (V', E')$ with edge capacities c'_e such that the following claim holds.

Claim 1. There exists a flow $f : E \rightarrow \mathbb{R}_{\geq 0}$ in G satisfying edge capacity constraints $f_e \leq c_e$ and the additional constraints given by equation (1) if and only if there is a flow $f' : E' \rightarrow \mathbb{R}_{\geq 0}$ of G' satisfying edge capacity constraints $f'_e \leq c'_e$ with equal value.

For this problem, make sure to (1) clearly state your construction of G' , and (2) prove that claim 1 holds.

Solution:

Lewis Arnsten

Start time: Thursday 3pm

Problem 1:

- 1) Our goal is to construct a directed graph $G' = (V', E')$ with edge capacities c'_e such that claim 1 holds:
 - a) For every vertex v in our graph G' we will create two vertices v_{in} and v_{out} (these will replace v).
 - b) We will create a new edge (v_{in}, v_{out}) with capacity D_v .
 - c) Finally, we convert all edges (u, v) in graph G to (u, v_{in}) in our graph G' . Likewise, we will convert every edge (v, w) in G to (v_{out}, w) in G' .
- 2) Now we must prove that claim 1 holds (should be clear from construction that solving maximum flow on this new network is equivalent to solving maximum flow with vertex capacities in original network):
 - a) We constructed our graph G' to split each node v into two nodes such that every incoming edge to v enters the node v_{in} and every outgoing edge from v exits v_{out} .
 - b) The capacity of our new edge (v_{in}, v_{out}) is D_v .
 - c) Given our construction of graph G' , if there exists a flow f satisfying the constraints in claim 1, there must exist an equivalent flow f' . This is true since, when considering how flow translates between G and G' :
 - i) $f(i, j) = f'(i_{in}, j_{out})$
 - ii) $f'(i_{in}, i_{out}) = f(i, x)$ for a certain x
 - iii) $f(y_{in}, y_{out}) = f(x, y)$ for a certain x
 - d) Additionally, if there exists a flow f' in G' , by the same process described in c an equivalent flow f can be found in G .

PROBLEM 2 (Hacking Computer Networks - 25 points). A hacker is trying to split a computer network by disconnecting as few pairs of computers as possible. The hacker writes out the network as a graph where vertices represent computers in the network and edges represent connections between pairs of computers. The hacker would then like to compute the minimum value k such that there exists k edges whose removal disconnects the graph into two or more connected components.

Given a connected, undirected graph $G = (V, E)$, provide an algorithm for computing k that runs in time bounded by a polynomial function of $|V| + |E|$. Clearly state your algorithm, prove that your algorithm is correct, and prove that it runs in polynomial time by providing an explicit $O(\cdot)$ bound.

Solution:

Problem 2:

Our goal in this problem is to find the minimum value k such that there exists k edges whose removal disconnects the graph into separate connected components.

To do so we will construct our algorithm as follows:

We are given a undirected graph $G = (V, E)$ from which we will select a vertex x

We will remove x from the vertices V

$k = \infty$

For all vertices v in V :

Create a graph G_{xv} (G_{xv} is an $s \rightarrow t$ directed graph where $s = x$ and $t = v$)

Use Edmonds-Karp to find the maximum flow f on G_{xv}

If $k > f$:

$k = f$

Return k

Time complexity: The main loop for this algorithm will execute $V-1$ times (once for every vertex excluding x). Every time it executes we will run Edmonds-Karp. Thus the complexity of this algorithm is equal to that of Edmonds-Karp, $O(|V||E|^2) * O(V) = O(|V|^2|E|^2)$.

Correctness: Given a $x \rightarrow y$ flow network G_{xy} connecting any two vertices (other than x and itself), Edmonds-Karp will find the maximum flow. Our algorithm runs for every vertex v (other than $x=v$) in our undirected graph, calculating the maximum flow from x to v . The value k returned by this algorithm will thus be equal to the minimum of the maximum flows. We know that the maximum flow between two vertices is equal to the total capacity of the edges in a minimum cut. Thus the value k computed by this algorithm is the minimum value such that removing k edges disconnects the graph G into separate components.

PROBLEM 3 (Prestigious Jobs - 25 points). You are a flashy business spending and gaining money, and accumulating prestige. You are faced with a sequence of $n > 0$ job requests $j = 1, \dots, n$ where job j is to be completed on day j . For every job, you are allowed to **ACCEPT**, or **REJECT** the job, but you *cannot change the order* in which the jobs must be done (i.e. job j must be done before j' if $j < j'$). Every job j is also associated with a pair of integers m_j, r_j where $m_j \in \mathbb{Z}$ is the net gain in money received in accepting job j (positive or negative as some jobs may not be profitable), and $r_j > 0$ is the amount of prestige you gain from accepting the job.

Starting with $D = 0$ dollars, your goal is to choose a subsequence of the jobs that maximize total prestige gained, under the constraint that you *never go into debt*. That is, the amount of money you have needs to be non-negative as you progressively accept each job in the subsequence. Clearly describe an algorithm that outputs an optimal subset of jobs, running in time polynomial in $(n + M + R)$, where

$$M = \sum_{1 \leq i \leq n} |m_i|, \quad R = \sum_{1 \leq i \leq n} r_i.$$

That is, the runtime should be at most $O((n + M + R)^c)$ for some constant $c > 0$. Prove that your algorithm is correct, and runs in polynomial time by stating an explicit $O(\cdot)$ bound.

Solution:

Problem 3:

This problem is similar to the knapsack problem (which was itself an extension of subset sum). In the knapsack problem our goal was to find a subset S of maximum value $\sum_{i \in S} V_i$, subject to the restriction that the total weight of the set should not exceed W : $\sum_{i \in S} W_i \leq W$.

To devise an algorithm which maximizes the total prestige gained we must first find the maximum prestige value for every subset of jobs and every possible cost value greater than 0 (we will call this value $OPT(k, M)$). To do so we will maintain a two-dimensional array $T[i, w]$ where the rows represent each subset of jobs and the columns represent every possible cost value.

Algorithm 1:

This algorithm will keep track of two two-dimensional arrays maintained within nested for loops. The first two dimensional array T will keep track of the maximum achievable prestige for every possible subset of jobs (rows) and every possible cost greater than or equal to 0 (columns) (this array is similar to the knapsack problem). The second two dimensional array C will keep track of the exact cost values for each element in T .

When considering any value OPT_k , we must know the best solution we can get using a subset of the first $k-1$ jobs subject to the restriction that we never go into debt. We will thus consider subproblems for each initial set of $\{1, \dots, k\}$ jobs, and each possible value for the remaining available money. We know that jobs $k = 1, \dots, n$ have associated costs m_k . If we examine the costs of all the jobs, and sum all the jobs with positive money values, we can find the maximum possible sum of costs M_{max} . We can then construct subproblems for each $k=0, \dots, n$ and each cost M greater than or equal to 0 and less than or equal to M_{max} . Thus, we will use $OPT(k, M)$ (this is a value in our two-dimensional array T) to indicate the value of the optimal solution using a subset of jobs $\{1, \dots, k\}$ achieving a cost M . Similar to the subset sum problem, we can now express the following recurrence for finding a value $OPT(k, M)$ in terms of values from smaller problems:

- Consider the column M in our array T (here referred to as OPT). When considering the k th job (k th row), if the cost of completing that job m_k is greater than M , then $OPT(k, M) = OPT(k-1, M)$. Additionally, if the cost of completing that job k is less than -(maximum value in $C[k-1, :]$), then $OPT(k, M) = OPT(k-1, M)$.
- Now, consider the case where the k th job has a cost greater than or equal to 0, but less than M . Here we can define $OPT(k, M)$ using $OPT(k, M) = \max(OPT(k-1, M), r_k + OPT(k-1, M - M_k))$.
- Finally, consider the case where the k th job has a cost less than 0 but greater than $-(C[k-1, M_i]$ for some i). Here we can also define $OPT(k, M)$ using $OPT(k, M) = \max(OPT(k-1, M), r_k + OPT(k-1, M - M_k))$.

Jobs \leftarrow array of values $1, \dots, n$ where each job $jobs[n] = (m_n, r_n)$

Array $T[0 \dots n, 0 \dots W]$

Array $C[0 \dots n, 0 \dots W]$

Initialize $T[0, M] = 0$ for each $m=0, 1, \dots, M$

Initialize $C[0, M] = 0$ for each $m=0, 1, \dots, M$

```

For i=1,2,...,n
  For m=0,...,M
    //Use the recurrence above to compute T[i, m]
    If jobs[i] [1] > m or jobs[i] [1] < -max(C[i-1, :]):
      T[i,m] = T[i-1,M]
    Else If jobs[i] [0] >= 0:
      T[i,m] = max(T[i-1,m], jobs[i][1] + T(i-1,m-jobs[i][0]))
    Compute corresponding C[i, m]

Return T [n, M]

```

Correctness: This algorithm differs from the subset sum and knapsack problems in that a certain cost value for a given subset of jobs can be achieved from both above and below. In other words, when finding the optimum prestige value for the subset of jobs 1,..,k for a given cost value M, if the cost value of the job k is positive, then we must ensure that the previous cost value we use is low enough to achieve the value M. Conversely, if the cost value of the job k is negative, then we must ensure that the previous cost value we use is high enough to achieve the value M. The construction of this algorithm ensures that $OPT(k,M) = T[n,M] =$ maximum achievable prestige for every possible subset of jobs and every possible cost greater than or equal to 0. Thus, using induction we can prove that each value $T[n,M]$ is the optimal prestige for jobs 1,..,n with cost M.

Time complexity: This algorithm must compute all values $OPT(k,M)$ in the two dimensional arrays T and C. This is done by checking n jobs * M , where $M =$ maximum possible sum of the costs of all jobs (this can be computed in $O(n)$ and is at most $\sum |m_i|$ for all $i \leq n$). Thus, a reasonable upper bound for this algorithm is $O(n*M)$.

PROBLEM 4 (Banquets - 25 points). Sometime in the not-too-distant future, friends gather for a banquet-style dinner at a large restaurant. There are n people and n dishes. (Again: we expect the same number of people and dishes!) Every participant only likes certain dishes. The goal is to seat the n people around a circular table with a dish between every pair of participants, so that every participant likes both dishes placed at their sides.

Input: An $n \times n$ binary matrix M such that $M_{ij} = 1$ if person i likes dish j and $M_{ij} = 0$ otherwise.

Decide: Whether there is an arrangement of all the people and dishes around a circular table, with a dish between any two adjacent people, such that every person likes both dishes placed at their two sides.

Note: every dish comes in a single plate and can be only placed in a single position. Show that this problem is NP-complete. (The banquet is still a success.) Hint: at least one of the following known NP-complete problems is a good choice to make use of in your proof: 3-SAT, Hamiltonian Cycle in undirected graphs, 3-Coloring. (The other two may be less convenient, so choose carefully).

Solution:

Problem 4:

We will approach this problem (henceforth referred to as the *table-organization problem*) as a reduction from hamiltonian cycle in undirected graphs. To decide if there exists an arrangement of all the people and dishes around a circular table--with dishes between every two people--such that every person likes both dishes placed at their two sides, we will consider a graph G.

Table-organization problem is in NP: To find a solution to the table-organization problem we must simply find a cycle in G such that every person is located next to two dishes that they like. It is clear that this can be done in polynomial time.

Table-organization problem is NP-Hard: We will show that some NP-complete L is polytime-reducible to Table-organization. In this case we choose L = Hamiltonian Cycle in an undirected graph.

Reduction: Given a Graph G we construct our seating arrangement as follows: There are n nodes in G. Let these n nodes represent both people and dishes. In other words, every node v_i in G will represent a tuple, $v_i = (x_i, y_i)$, where x_i refers to the person and y_i refers to the dish. Consider a node v_i in G representing a person. This node is connected to a set of nodes V, representing dishes. Let the person like all the dishes in V as well as the dish their node represents ($M_{ii} = 1$). This means that, for each v_i in G, for all nodes $1, \dots, j$ in the corresponding V, x_i likes y_j ($M_{ij} = 1$).

Claim: The existence of a hamiltonian cycle in graph G will necessarily imply a solution to the table-organization problem. Let there exist a hamiltonian path H in G. Now we will choose a node v_i on the hamiltonian path H. This is where we will start our solution to the table-organization problem. Since a hamiltonian path/cycle visits every node exactly once, there exists an edge between v_i and another node v_{i+1} . In our solution to the table-organization problem we will place x_i (person i) next to y_{i+1} (dish $i+1$). We know that this is valid because the existence of an edge between v_i and v_{i+1} implies that person x_i likes dish y_{i+1} . We can continue to construct our solution placing x_{i+1} next to y_{i+1} (again we know this to be valid as every person likes the dish represented by their node). Next, by the definition of a hamiltonian cycle, we know that there exists an edge in H connecting v_{i+1} to just one other node v_{i+2} . We will continue our solution by placing y_{i+2} next to x_{i+1} and so on until we reach the last vertex on H, which is connected to v_i by an edge.

Claim: Conversely, if there exists a valid solution to the table-organization problem, then there must exist a hamiltonian cycle in G. Given a valid solution to the table-organization problem, $x_1, y_1, x_2, y_2, x_3, \dots, x_n, y_n$, the corresponding cycle $(v_1, v_2), (v_2, v_3), \dots, (v_n, v_1)$ must be a hamiltonian path. This is true because: We know that every node v_i is visited once. Additionally, since in a valid solution every person is seated next to two dishes they like and is connected to those dishes by an edge, every node v_i is connected to neighboring nodes v_{i+1} and v_{i-1} by an edge. Thus, the

cycle corresponding to the solution to the table-organization problem must be a hamiltonian cycle.

It follows from the above that hamiltonian cycle in undirected graphs polytime reduces to the table-organization problem, which is therefore NP-hard. We have completed the proof that the table-organization problem is NP-complete. Thus, we can decide whether there exists an arrangement of all the people and dishes around a circular table, with a dish between any two adjacent people, such that every person likes both dishes placed at their two sides.