CompSci 330 Design and Analysis of Algorithms
Homework 8, Fall 2024 Duke University

TODO: Add your name(s) here

Due Date: November 18, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on Gradescope. You can use the internet, but looking up solutions or using LLMs is unlikely to help you prepare for exams. See the homework webpage for more details.

Grading. Theory problems will be graded by TAs on an S/I/U scale. Applied problems typically have a separate autograder where you can see your score. See the course policy webpage for details about homework grading.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Flow Cycles). Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges. Let s, t be two distinct vertices in G, and let $c : E \to \mathbb{Z}_{\geq 0}$ be an edge capacity function such that each edge capacity is a non-negative integer, i.e., $c(u \to v) \geq 0$ is the (integer) capacity of the edge $u \to v$. Let $f : E \to \mathbb{Z}_{\geq 0}$ be a (s, t)-flow function in G in which one of the edges $v \to s \in E$ entering the source vertex s has $f(v \to s) = 1$. Assume that f has non-negative flow value.

(a) Prove that there must exist another (s, t)-flow $f' : E \to \mathbb{Z}_{\geq 0}$ with $f'(v \to s) = 0$ and $|f| = |f'|$ (that is, having the same (s, t)-flow value).

Solution 2a. We argue that there must be a cycle $C = s \to u_1 \to u_2 \to \ldots \to u_k \to v \to s$ where $v \to s$ is the edge with $f(v \to s) = 1$ such that $f(e) \geq 1$ for every edge in C. Since $f(v \to s) = 1$ and flows are nonnegative integers, the flow conservation constraint on v implies that there must be an edge $u_k \to v$ with $f(u_k \to v) \geq 1$. The same reasoning holds for u_k, then for u_{k-1}, and so forth inductively, until we reach the start vertex s that does not have a flow conservation constraint.

Given that C exists, consider the flow f' defined as

$$f'(e) == \begin{cases} f(e) - 1 & \text{if } e \in C \\ f(e) & \text{otherwise.} \end{cases}$$

Note that f' has $f'(v \to s) = 0$ since $f(v \to s) = 1$ and $v \to s \in C$. We also argue that f' is also a valid flow. $f'(e) \leq f(e)\ \forall e \in E$ so f' cannot violate capacity constraints if f was a valid flow. Furthermore, for any $u \in C$, the net flow into and out of u are both reduced by 1, whereas for $u \notin C$ neither in flow nor out flow are changed. In either case, the flow conservation constraints at f' are still satisfied since they were satisfied for f. Finally, note that by the same logic the net flow out of s has not changed, so $|f'| = |f|$.

(b) Given f, describe an $O(m)$ runtime algorithm to compute f'. Briefly explain why the algorithm is correct, referencing the proof of existence from the previous part. Analyze the runtime of your algorithm.

Solution 2b. Run a single BFS starting from s and only considering edges $u \to v$ that have $f(u \to v) \geq 1$. As argued previously, there must be a cycle C beginning from s, which can be identified by identifying a node reachable from s with an edge to s. C is the path from s to this node in the BFS tree plus the edge back to s. Compute f' by enumerating the edges and setting $f'(e) = f(e) - 1$ for all $e \in C$, and $f'(e) = f(e)$ otherwise.

The runtime complexity for the DFS on a subset of the edges is $O(|E|)$. The cycle is identified via BFS in $O(|E|)$ time. Finally, computing f' from f requires enumerating the edges, $O(|E|)$ time.

Problem 2 (New Curriculum). The Trinity Curriculum Development Committee (TCDC) recently submitted a new curriculum, which was approved by the Arts & Sciences Council. The new curriculum requires a student to take a number of classes r_i from m different categories G_i in order to graduate. There are n total classes. **A taken class counts towards at most one requirement**; if a class is in two or more categories, it must be decided which one, if any, that the class counts towards.

For example, suppose there are $n = 5$ classes A, B, C, D, E and $m = 2$ categories:

- You must take (at least) $r_1 = 2$ classes for category $G_1 = \{A, B, C\}$.
- You must take (at least) $r_2 = 2$ classes for category $G_2 = \{C, D, E\}$.

Then a student who has only taken courses B, C, D cannot graduate (since C cannot be put towards both categories), but a student who has taken either A, B, C, D or B, C, D, E can graduate.

Describe and analyze an algorithm to determine whether a given student can graduate. The input to your algorithm is the list of the m categories (each specifying a subset G_i of the n courses and the number of courses r_i that must be taken from that subset) and the list of courses that the student has taken. Justify the correctness of your reduction and analyze the runtime.

Solution 2. We reduce the problem to maximum flows, as follows. Let L be the list of courses taken by the student for which we wish to determine if they can graduate. Let $G = (V, E)$ be a flow network with edge capacities $c : E \to \mathbb{Z}^+$, where V consists of source and sink vertices s and t, respectively, plus each class C_i and each category G_j. Thus $|V| = n + m + 2 = O(n + m)$. For each class $C_i \in L$ taken by the student, we add the directed edge $C_i \to G_j$ with capacity 1 for each category G_j that includes C_i, i.e., for all categories in $\{G_j \mid C_i \in G_j\}$. Finally, we add directed edges from s to each course C_i with capacity 1, and directed edges from G_j to t with capacity r_j, the number of courses from G_j required in order to graduate. Then $|E| \le nm+n+m = O(nm)$.

Let $R = \sum_j r_j$ be the sum of all requirements. By construction, the maximum flow in G is bounded by the sum of capacities out of s, which is n, as well as the sum of capacities into t, which is R. We claim there is a flow in G of value R if and only if the student can graduate.

- $\Rightarrow$: Suppose there is a flow f in G with $|f| \ge R$. As argued above, the sum of capacities into t is R, so all incoming edges to t are saturated in f. A flow decomposition P of f into paths consists of weight-1 paths since all capacities of edges leaving s are 1. Every path includes (exactly) one edge $C_i \to G_j$ between a course C_i and category G_j, which exists if and only if $C_i \in G_j$ and $C_i \in L$. Thus we assign course C_i to satisfy one of the r_j courses from G_j. Assigning courses in this way indeed results in a way for the student to satisfy all requirements, as each course is assigned exactly once because any course is included in at most one path of the decomposition (as its incoming capacity is 1).
- $\Leftarrow$: Suppose the student can graduate, so there is a subset of taken classes $L_j \subset C_j$ with $|L_j| = r_j$ that are used to satisfy the requirements of C_j (and no other category). If there are more than r_j classes used to satisfy the category, we pick L_j as any r_j such classes. For each category G_j, we push a unit of flow on the path $s \to C_i \to G_j \to t$ for each taken course $C_i \in G_j$ being used to satisfy the r_j required courses from C_j. Pushing flow in this way satisfies all capacities of the edges since the edges exist, any class is used to satisfy at most one category requirement, and any category has at most r_j courses to satisfy it.

Constructing G takes $O(V + E) = O(nm)$ time, and computing a max-flow in G takes $O(E|f^*|)$ time via Ford-Fulkerson. We have that $|f| \leq \min\{n, R\}$, so the runtime is $O(ER) = O(nmR)$ if $R \leq n$, otherwise it is $O(En) = O(n^2m)$ which matches the runtime $O(EV) = O(n^2m)$ of Orlin's algorithm.

A reduction to maximum matching: Instead of explicitly reducing to max-flow, we could create the following undirected bipartite graph $G = (\mathscr{C} \sqcup \mathscr{G}, E)$ where $\mathscr{C}$ is the set of courses, $\mathscr{G}$ is the set $\bigcup_j G_j \times \{1, \ldots, r_j\}$; that is, there are r_j copies of category G_j as vertices in $\mathscr{G}$. E contains an edge between $C_i \in \mathscr{C}$ and vertex $(G_j, x) \in \mathscr{G}$ if and only if the student has taken course C_i and G_j includes the course. Then one can show there is a matching of size at least $R = \sum_j r_j$ if and only if the student can graduate. Constructing G takes $O(nm)$ time and computing a maximum matching takes $O(nmR)$ time.

Problem 3 (Matrix Rounding) Suppose we are given an array $A[1..m][1..n]$ of non-negative numbers that are **not** integers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.3 & 3.3 & 2.4 \\ 3.8 & 4.1 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

a. Describe and analyze an efficient reduction-based algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible. Justify that your algorithm is correct, and analyze the runtime. *[Hint: There is a correct reduction involving a flow network with one vertex per row and one vertex per column, plus $O(1)$ more total.]*

b. Justify that a matrix can be rounded if and only if the sum of entries of each column is an integer and the sum of entries of each row is an integer. Note that this result is not an immediate consequence of part (a)! Part (a) states it is possible to compute a solution when one exists, whereas this part asks you to characterize exactly when they do exist (and thus for what kind of inputs your algorithm will report one).

*[Hint: Consider a particular **fractional** flow on your graph. Recall properties of flow networks with integer capacities.*

a. We reduce the problem to maximum flows, as follows.

Let B be the $m \times n$ matrix where $B[i,j] = A[i,j] - \lfloor A[i,j] \rfloor$; that is, B contains the fractional part of each element in A. It can be verified that A can be rounded if and only if B can be rounded, as rounding only affects the fractional parts of the elements in A.[2]

For each row r_i of B, let $w(r_i) = \sum_{1 \le j \le n} B[i,j]$ be the sum of elements in r_i. Similarly, for each column c_j of B, let $w(c_j) = \sum_{1 \le i \le m} B[i,j]$ be the sum of elements in c_j. Finally, let $W = \sum_i w(r_i) = \sum_j w(c_j)$ be the sum of all elements in B.

First, we handle an obvious case directly: if there exists a column c_j where the sum of its entries is not an integer or a row where the sum of its entries is not an integer, then we report IMPOSSIBLE since any rounding of B yields integral row and column sums. Otherwise, we proceed by constructing a flow network.[3]

Let $G = (V, E)$ be a flow network with edge capacities $c : E \rightarrow \mathbb{Z}^+$, where V consists of source and sink vertices s and t, respectively, plus a vertex r_i for each of the m rows of B and a vertex c_j for each of the n columns of B. Thus $|V| = n + m + 2 = O(n + m)$. For each element $B[i,j]$ in B we add the directed edge $r_i \rightarrow c_j$ with capacity $\lceil B[i,j] \rceil$, which is $1 > B[i,j]$. Finally, we add directed edges $s \rightarrow r_i$ with capacity $w(r_i)$ for each row r_i and add directed edges $c_j \rightarrow t$ with capacity $w(c_j)$ for each column c_j. Note that G has only integral capacities: those out of s and into t are integral by the above case, and the capacities between row and column.

[2] Assuming A only has elements between 0 and 1 is helpful in one direction of the proof; if one uses the construction with A instead of B, then it is correct but harder to prove it.

[3] As we later prove in part b, it would be correct at this point to report POSSIBLE, but proving b requires the rest of our reduction, so we proceed. To obtain the rounding of B itself, not only report whether B can be rounded, performing the reduction in full is needed.

We claim that G has max-flow value at least W if and only if B can be rounded.

Indeed, if B can be rounded to matrix B', we put $B'[i,j] \leq \lceil A[i,j] \rceil$ flow on edge $r_i \to c_j$ for each i, j. The sums of columns and sums of rows in B' are the same as those in B, so putting $w(r_i)$ flow on edges $s \to r_i$ and putting $w(c_j)$ flow on edges $c_j \to t$ makes the resulting flow satisfy conservation. Furthermore, the value is clearly W as the edges leaving s are saturated (and the edges into t are saturated) with total capacity W.

Next, suppose G has a max-flow f with value W, which we assume to be integral as G has integral capacities. f saturates all edges out of s and all edges into t. Then the flow in and out of r_i is $w(r_i)$ and the flow in and out of c_j is $w(c_j)$. Next, see that $\lfloor B[i,j] \rfloor = 0$ since all elements of B are less than 1. Since f is feasible, we have $\lfloor B[i,j] \rfloor \leq f(r_i \to c_j) \leq \lceil B[i,j] \rceil$ for all i, j, where the second inequality follows from the capacities of the edges.[4] Furthermore, we have that $f(r_i \to c_j)$ is equal to either $\lceil B[i,j] \rceil$ or $\lfloor B[i,j] \rfloor$ since f is integral and these values are consecutive integers. Thus, the matrix $B'[1..m, 1..n]$ of B where $B'[i,j] = f(r_i \to c_j)$ is a rounding of B.

We conclude the runtime analysis. Constructing B takes $O(n+m)$ time. G can be constructed in $O(E+V) = O(nm)$ time. Since $W \leq mn$, which is the capacity of the edges leaving s (entering t), the max-flow is at most mn. Thus Ford-Fulkerson takes $O(Emn) = O(m^2n^2)$ time. On the other hand, Orlin's algorithm has runtime $O(VE) = O((n+m)mn)$. Thus we use Orlin's algorithm whose runtime dominates the overall runtime: $O((n+m)mn)$.

A common missed detail: Consider the matrix A below:

$$\begin{bmatrix} 2.5 & 2.5 & 2.5 & 2.5 \\ 2.5 & 0.5 & 0.5 & 2.5 \\ 2.5 & 0.5 & 0.5 & 2.5 \\ 2.5 & 2.5 & 2.5 & 2.5 \end{bmatrix}$$

If we consider the flow network obtained by placing $\lceil A[i,j] \rceil$ capacity on edge $r_i \to c_j$, compute a max-flow, then "round" by taking the (integer) flow on these edges to be the new entries of the matrix we could receive the following matrix: This is not a valid rounding of the initial

$$\begin{bmatrix} 3 & 3 & 3 & 1 \\ 3 & 0 & 0 & 3 \\ 3 & 0 & 0 & 3 \\ 1 & 3 & 3 & 3 \end{bmatrix}$$

A. However, it can be shown there exists at least one max-flow whose flow values across the entries correspond to a valid rounding.

b. In the previous part, we argued that if A has a sum of columns or sum of rows which is not an integer, there is no way to round A. Thus it remains to prove that if A indeed has its sums of columns as integers and sums of rows as integers, then there is indeed a way to round A. Consider the flow network G and matrix B constructed in the previous part. We consider the (fractional) flow f obtained by pushing $B[i,j]$ flow along the path $s \to r_i \to c_j \to t$. This is feasible and has flow equal to W, the sum of all elements in B. Thus the max-flow in B has

[4] This is where it is easier to work with B instead of A; see the end of this solution for a troublesome input matrix A whose max-flow may not correspond to a valid rounding.

value at least W. Furthermore, the max-flow value is at most B (as it is the sum of capacities out of s and the sum of capacities into t). So f is a max-flow. As argued above, there is a rounding of B if the max-flow is W, so B (and thus A) can be rounded.

Problem 4 (Applied) You are a city planner trying to optimize traffic flow in the city's transportation network. Imagine a city with a complex network of roads and highways but with only one entry point s and one exit point t. You are asked to increase the traffic capacity driving from s to t but are only allotted with the money to widen one road such that its capacity would increase.

Assume now the city road map is converted into a directed graph (not necessarily acyclic) with nodes labeled as integers and with non-negative integer capacities on the edges. Your function will take in two parallel lists edges and capacities, where each edge is a tuple. For example, $(2, 4)$ is an edge that goes from starting node 2 to destination node 4. You are also given the label of a source vertex and a target vertex.

We say that an edge is a *priority edge* for routing traffic flow from source s to target t if increasing the capacity on that edge by 1 (with no other changes to the graph) would increase the value of the maximum flow from s to t.

Given the above inputs, **you should design and implement an algorithm that returns a list of *all* priority edges in the graph (or an empty list if there are none)**. The list can be in any order. For full credit, your solution will need to have an empirical runtime that is within constant factors of running Ford-Fulkerson using BFS.[5]

[Hint: Recall the correspondence between the value of a maximum (s, t)-flow and the capacity of a minimum (s, t)-cut.

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline.

- **Python.** You should submit a file called `flow.py` to the Gradescope item "Assignment 6 - Applied (Python)." The file should define (at least) a top level function `find_edges` that looks like:
  - `def find_edges(edges:[(u:int,v:int)], capacities:[int], s:int, t:int)`

  and returns a list of tuples `(u,v)` that are priority edges or an empty list `[]`
- **Java.** You should submit a file called `Flow.java` to the Gradescope item "Assignment 6 - Applied (Java)." The file should define (at least) a top level function `findEdges` that looks like:
  - `public List<int[]> findEdges(int[][] edges, int[] capacities, int s, int t)`

  where `edges` is a 2D array where `edges[i]` is an edge from `edges[i][0]` to `edges[i][1]` and `capacities[i]` is the capacity of `edges[i]`. Return either a list of edges or an empty list `[]`

Solution 4.

- Compute the maximum $s - t$ flow f in G once using Ford-Fulkerson with BFS to find augmenting paths. Let the resulting residual graph computed as part of the algorithm be G_f.
- Compute the residual graph G_f in $O(n + m)$ time and its reversal, G_f^{rev}.

[5] Picking the shortest augmenting path in Ford-Fulkerson is known as the Edmonds-Karp algorithm, who showed it runs in $O(m \cdot \min\{|f^*|, n^2\})$ time.

- The priority edges are $u \to v \in E$ such that u is reachable from s in G_f and v is reachable from t in G^{rev}, which can be identified with two BFS calls.

The runtime of Ford-Fulkerson is $O(m|f^*|)$, and the two BFS calls takes $O(m)$ time. The overall runtime is $O(m|f^*|)$.