CIS 320 Problem Set 4

Due: Wednesday 11/10/21

Welcome to Problem Set 4, which will be a unifying journey through the main topics covered so far: DP, greedy algorithms, and flows. A few notes before you begin.

- You may collaborate with up to 4 other people. All work must be written up individually. You may collaborate, but you cannot copy. E.g. you can talk to your friends about ideas for a problem together, but you cannot copy their solution and just change some words around. **List your collaborators at the top of your submission**.
- Googling or looking up solutions in any way is not allowed.
- Cheating is not worth it! Your grade in this course does not define your worth as a person, and in 10 years you will not care about your bad grade on a homework assignment. But you will care if you are caught plagiarizing: plagiarizing has serious consequences, including the potential of expulsion.
- These problems are designed to challenge you. Start them early; if we've done our job well writing them, you will have to chew on them for a while before finding the solutions, and that means you will do best if you can sleep on your solutions rather than starting them the night before the due date.
- You do not need to implement anything in code. In fact, please do not. Pseudocode or a clear English explanation of your algorithm are both acceptable: in some cases pseudocode may be clearer than plain English, and in others the plain English might be better.
- If a question asks you to come up with an algorithm with a certain target runtime (say $O(n \log n)$), and you don't know how to achieve this runtime but know how to solve the problem less efficiently (in, say, $\Theta(n^2)$ time), write that down! Depending on how inefficient your solution is compared to the benchmark, it will receive a varying amount of credit.
- For each of our algorithm design questions, you should come up with a *deterministic* algorithm unless the question specifies that a randomized algorithm is wanted.
- All analysis must be mathematically rigorous. Any answer you provide should be proven.
- Remember, we can't evaluate your work if we can't understand it. Communicating mathematical and/or complex ideas is an important skill in computer science; treat your problem sets as practice.
- You should use LaTeX to typeset your solutions.
- You do not need any knowledge of Lord of the Rings in order to complete this problem set. Note that the majority of the TAs writing the problems did not have any prior experience with Lord of the Rings. Ira has enough Lord of the Rings knowledge for everyone combined.
- Have fun!!

A Faceoff: Flows vs. Dynamic Programming

After the immense popularity of the chess-themed show *The Queen's Gambit*, streaming platform Nutflex follows the trend and releases a new show, *The Dominoes Cohabit*. As a result, the ancient game of Dominoes is back to the zenith of its popularity (as is a certain pizza restaurant chain). We have decided to use this opportunity and treat you to two dominoes-themed problems.

Problem 1 (Go with the Flow)**.** Your friend Steve (yes, the one who helped evaluate the quality of your poetry) visits your place for another poetry-storming session. At this point he has not slept for 96 hours, so in addition to crazy poetic ideas his personality has developed a touch of violence. While you are preparing tea with honey in the kitchen, Steve grabs your favorite chess board of size $m \times n$, and knocks out some of the squares, so that now your chessboard has holes instead of squares in certain locations! He then realizes that you will not like the sight of a destroyed chessboard, and decides to appease you by tiling the remnants of the board with as many 2×1 dominoes as possible! Dominoes can be oriented vertically or horizontally, and are not allowed to overlap, nor cover any of the holes.

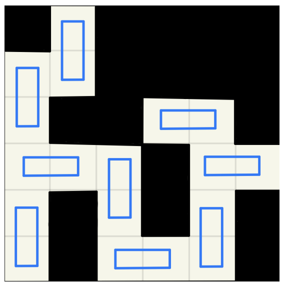

Figure 1: Holes are in black, and the remaining squares are properly tiled with dominoes.

Design a *flow-based* algorithm (i.e. an algorithm that reduces the problem to a max-flow problem, and then solves that using the Ford-Fulkerson algorithm from class) that takes as input m, n, and the locations of the holes (which can be anywhere on the board, and can come in any number), and outputs the maximum number of dominoes that you can tile the remaining board with.

Problem 2 (DP Strikes Back)**.** Given an $m \times n$ chessboard, and 2×1 dominoes, how many ways are there to completely tile the chessboard with dominoes without overlap? (That is, each square of the board must be covered by precisely one domino.) Assume all dominoes are identical, and can be used in both the vertical and the horizontal orientation.

Come up with, and analyze, a $O(nm \cdot 2^{\min\{n,m\}})$ runtime dynamic programming algorithm that, on input m, n outputs the number of domino tilings of the $m \times n$ board, as defined above. *Hint: We bet you have seen this runtime before... could it be on a previous homework?*

Flows for Machine Learning

Problem 3 (Pre-Neural-Networks Image Processing)**.** In year 2067, famous Convolutional Neural Networks Sensei Alex K finally decides to retire and move to her countryside house in rural Massachusetts. One morning, she flips through her favorite childhood photos and is overcome by a sense of nostalgia: she remembers how cool and simple life was before CNNs were invented, and challenges herself to process these images in an old-school fashion: instead of CNN, she wants to use her favorite algorithmic technique of *flows* that she learnt in her undergrad algorithms class.

To define the task to be carried out, she settles on foreground/background detection. Specifically, the task is to automatically split a photo (in the digital format) into *foreground* and *background* pixels.

Formally, a digital image is represented as an $m \times n$ matrix of pixels. Thus, each pixel is represented using its row and column indices as (x, y). Each pixel's neighbors (there are up to four of them) are defined to be the pixels that it shares a side with. We call $N_{m,n}$ the set of pairs of neighboring pixels. For each pixel (x, y), Alex has estimated, using statistical methods, two quantities: $f_{x,y} \geq 0$, which is the probability that (x, y) is a pixel in the foreground, and $b_{x,y} \geq 0$, which is the probability that (x, y) belongs to the background. Furthermore, for every two pixels $(x_1, y_1), (x_2, y_2)$ that are neighbors, Alex has estimated $s_{(x_1,y_1),(x_2,y_2)} \geq 0$, which is a measure of synergy between the two pixels; the higher $s_{(x_1,y_1),(x_2,y_2)}$, the more likely it is that both these pixels are of the same type (both foreground or both background).

Given these input data (the height and width m, n of the image and the estimated parameters — the collections of numbers $f_{x,y}, b_{x,y}$ and $s_{(x_1,y_1),(x_2,y_2)}$), Alex needs to partition the entire set of the image's mn pixels into the set F of foreground pixels and the set B of background pixels. The *loss* of any partition (F, B) is defined to be:

$$\ell(F, B) := -\sum_{(x,y)\in F} f_{x,y} - \sum_{(x,y)\in B} b_{x,y} + \sum_{\substack{\big((x_1,y_1),(x_2,y_2)\big)\in N_{m,n}: \\ \text{one pixel is in } F, \text{ the other in } B}} s_{(x_1,y_1),(x_2,y_2)}.$$

The first and second summation reward the placement of each pixel (x, y) into foreground or background in accordance with which of $f_{x,y}$ and $b_{x,y}$ is a larger number. The last term penalizes each pair of neighboring pixels that end up on the opposite sides of the partition, by the amount equal to their synergy.

Your task is to formulate and analyze a *flow-based* algorithm that finds a minimum-loss partition (F, B) of the set of pixels into foreground and background pixels. Specifically, Alex asks you to reduce this problem to finding a minimum cut in a directed network. In order to do that: 1) Define a weighted directed network $G(V, E)$ with a dedicated source and a sink, such that its cuts corresponds precisely to the partitions (F, B) of the pixel set, and such that the value of each cut is equal to the loss of the corresponding partition (F, B); 2) Show the correctness of this reduction, and analyze the runtime of the Ford-Fulkerson algorithm as applied to this setting.

The Ultimate Greed: It Ends Here[1]

Problem 4 (Matroids: An Introduction)**.** By this point, you might be wondering (are you?): is there a general way to characterize settings where the greedy approach provably results in optimal solutions? In this and the next problem, we will explore a general class of domains where greed is the way to go: *matroids.*

We begin with the definition. A pair $(E, \mathcal{I})$, where E is a finite set and $\mathcal{I}$ is a collection of subsets of E, is called a *matroid* if the following three axioms are satisfied:

1. The empty set belongs to the collection $\mathcal{I}$: that is, $\emptyset \in \mathcal{I}$.
2. If a subset of E belongs to the collection $\mathcal{I}$, then all of its subsets also belong to $\mathcal{I}$: that is, for any A, B such that $B \in \mathcal{I}$ and $A \subseteq B$, it also holds that $A \in \mathcal{I}$.
3. Any non-maximal-size subset in the collection $\mathcal{I}$ can be augmented with a single element from any larger set in $\mathcal{I}$ such that the augmented set is also in $\mathcal{I}$. Formally, for any $A, B \in \mathcal{I}$ such that $|B| > |A|$, there exists an element $x \in B - A$ such that $(A \cup \{x\}) \in \mathcal{I}$.

As a useful bit of terminology, each set in the collection $\mathcal{I}$ is called an *independent set.*

Whew. This was a lot to parse. But the main takeaways from this set of axioms are as follows: a matroid describes a collection of subsets of a certain ground set, called independent sets, such that all subsets of an independent set are also independent, and such that any independent set, unless it already has the maximum size among all other independent sets, can be grown into a slightly bigger independent set by adding to it some element of the ground set (in fact some element from any larger independent set).

Part 4A. It is time to get comfortable with this abstract-looking notion: let us work out an example! Show that if $E = \{a, b, c, d\}$, and $\mathcal{I} = \{\emptyset, a, b, c, d, ab, ac, ad, bc, bd, cd\}$, then $(E, \mathcal{I})$ is a matroid.

[1] It actually does not end here. Matroids have further generalizations compatible with greedy algorithms, such as greedoids and matroid embeddings.

Now we are ready to discuss the main algorithmic property of matroids, which is the reason we are studying them. Informally, this property states: greed always works on matroids!

Formally, let us define a *weighted matroid* as follows. A weighted matroid is a tuple $(E, \mathcal{I}, \{w\}_{x \in E})$ such that $(E, \mathcal{I})$ is a matroid and for each element $x \in E$, it is endowed with weight $w_x > 0$. By extension, the weight of a subset $S \subseteq E$ is defined as the sum of its elements' weights: $w_S := \sum_{x \in S} w_x$.

Now here is our algorithmic question of interest: Given a matroid $(E, \mathcal{I})$, how do we find a min-weight maximally sized independent set $S \in \mathcal{I}$? The following simple greedy approach turns out to be the answer.

The Matroid Greedy Algorithm: Start with $S = \emptyset$. While you can, do the following: from the set of elements $x \in E - S$ such that $(S \cup \{x\}) \in \mathcal{I}$, select the min-weight element x' and add it to your set S: that is, let $S \leftarrow S \cup \{x'\}$. If none of the elements in $E - S$ can be added to S without making it non-independent, terminate and return the current set S.

Part 4B. Prove that for any matroid $(E, \mathcal{I})$, the Matroid Greedy algorithm indeed returns a min-weight *maximal* independent set S (i.e. the min-weight independent set among those independent sets that cannot be grown into bigger independent sets).

As it turns out, not only is the Matroid Greedy the "right" algorithm to be used with matroids, but conversely, matroids are in some sense the "right" setting for the Matroid Greedy algorithm to produce optimal solutions.

Part 4C. Prove that for any pair $(E, \mathcal{I})$, where $\mathcal{I}$ satisfies conditions 1 and 2 but not condition 3 (from the above definition of matroids), there exists a set of positive weights $\{w_x\}_{x \in E}$ such that the Matroid Greedy algorithm does not return a min-weight maximal independent set from $\mathcal{I}$.

Part 4D. Prove that for any weighted matroid, the *maximization* version of the Matroid Greedy algorithm (i.e. where at each step, a max-weight, instead of min-weight, element is added to the current independent set) always outputs the max-weight independent set.

Problem 5 (Matroids: Applications)**.** Having defined matroids and investigated their main algorithmic property, you can now reap a well-deserved reward: You will now be able to show that two seemingly disparate algorithms that you have learned in this class are instances of the Matroid Greedy algorithm.

Part 5A: Kruskal's Algorithm For any graph $G(V, E)$, its *graphical matroid* is defined to be the tuple $(E, \mathcal{I})$, where $\mathcal{I}$ consists of all acyclic subsets $S \subseteq E$ of the edge set of G (i.e. if one were to remove from G all edges in $E - S$, the remaining graph would be acyclic). First, show that the graphical matroid is indeed a matroid. Then, demonstrate that Kruskal's algorithm for finding MST in a weighted graph $G(V, E)$ is an instance of the Matroid Greedy *minimization* algorithm, as applied to the weighted graphical matroid of G.

Note: You may now wonder: is Prim's algorithm also an instance of some generic greedy algorithm, or is it custom-made for the particular problem of finding MST? It turns out it is indeed an instance of a general greedy algorithm for *greedoids*, which further generalize matroids.

Part 5B: Unweighted Bipartite Matchings via Nice Paths Consider a bipartite graph $G(V, E)$ with $V = (L, R)$. The *transversal matroid* of G is the tuple $(L, \mathcal{I})$, where a subset of left-hand-side vertices $S \subseteq L$ belongs to $\mathcal{I}$ iff there exists a matching M_S such that all vertices in S are matched in M_S.

Now, recall Problem 5 from Problem Set 3. There, you analyzed an algorithm for finding maximum-size matchings in unweighted bipartite graphs, which, at each step, increases the size of a tentative matching M_i by 1 by flipping edges belonging/not belonging to M_i along an M_i-nice path (if such a path exists).

First, prove that the transversal matroid is indeed a matroid. As a hint, begin by considering any two matchings M_1, M_2 with $|M_1| > |M_2|$, and building an M_2-nice path starting in some vertex $v \in L$ which is covered by M_1 but not M_2. Then, show that the algorithm from Problem 5 on Problem Set 3 is just the Matroid Greedy *maximization* algorithm with respect to the transversal matroid with all weights set to 1.