Homework 8: P/NP, quarter wrap-up

Due Date: The written parts of this assignment (problems 1-8) will be due at 11:59 PM on **Friday** December 9th. As always, that means that the TAs will grade **every** written problem you submit to gradescope before that deadline. You will also be able to resubmit two written problems with every homework; the resubmission for this week is due on **Monday** December 12th.

The coding problems for the entire quarter are due Friday Dec. 9.

Collaboration: You are allowed (and encouraged!) to discuss these problems at a high level with others. But, please read the full collaboration policy to ensure you are keeping your discussions at the right level, and remember to cite any collaborators.

Problems to Submit: We will count your 2 best mechanical question and your 5 best long-form questions.
We strongly recommend you skim all the problems in a section before attempting them. While we try to make problems in a given section of approximately equal difficulty, you may find some to be easier than others.

Note that we set the grade guarantees expecting you to have 1 mechanical and 3 long-form problems per assignment, so submitting only that would let you "keep pace," but as this is the last assignment, we wanted to give you a chance to catch up from earlier in the quarter.

Directions Unless otherwise noted, you are allowed to use algorithms described in class (or prerequisite courses) as though you had library implementations of them. For example, you can say "run the BFS-based 2-coloring algorithm from class on the graph G" or "run the bipartite checking algorithm from lecture 5 on G." We also have a list of data structures and algorithms you can use from 373 here.

Since this is a course about efficient algorithms, algorithms that are slower than necessary are unlikely to get "E" scores, but (unless otherwise noted) we do not care about constant factors. In general, an algorithm that is correct but (mildly) slower than optimal will get an equal or higher score to an algorithm that is fundamentally incorrect, but fast.

Mechanical Problems

1. True or False

For each of the following, you should answer either "known to be true", "known to be false", or "unknown" ("unknown" will be correct if and only if answering the question would resolve P vs. NP). Additionally, add 1-2 sentences of explanation.
We'll give 'E' for getting 4/5 (with explanation) and 'S' for 3/5 (with explanation).

(a) When trying to show our problem A is NP-hard, we take a problem B we already know is NP-hard, and reduce from A to B.

(b) NP-complete problems are a subset of NP-problems (that is, if A is NP-complete, then A is also in NP).

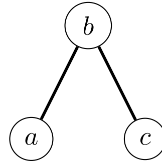
(c) NP problems are a subset of P problems (that is, if A is in NP, then A is also in P).

(d) The Longest Increasing Subsequence problem from lecture (given a sequence, return the length of the longest increasing subsequence) is in NP.

(e) Given a problem in NP has a linear time solution, all other problems in NP can be solved in linear time.

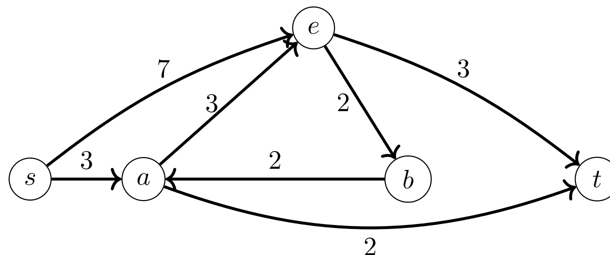
2. Mechanical Reduction

Given the graph below, transform this 3-coloring instance to a 3-SAT instance. Your solution should be a list of constraints on each 3-SAT variable. Make sure you include both edge constraints and consistency constraints. Refer to slide 42-50 in the reduction lecture for how to get started.



3. Linear Flow-Gramming

You are given the following graph where s is the source vertex, t is the sink vertex, and each edge is labeled with its capacity.



(a) Formulate a linear program for finding max flow on *this particular* graph. For this problem, we want you to give it in as much explicit detail as possible, so **do not use sigma/summation notation**. Describe what decision variables you will use, what your objective function is, and all constraints that you employ. Briefly explain the purpose of each. (Refer to Lecture 20, Slide 55 for how to get started.)

Long-Form Problems

4. Written: Reduction I

On Homework 6, you had a problem of assigning labels $0, 1, 2$ to employees at a company based on their direct reporting (tree) structure. (Make sure to remember this is an **undirected** graph)

Your boss was so impressed with the algorithm you wrote, they decided they want to sell your code to other companies for planning their vacations. There's only one catch, other companies have more complicated (not necessarily tree) reporting structures. Some people might have more than one supervisor, for example. But that should be an easy fix — you can handle that right?

The problem is now the following:

Input: A number n of employees (they will be numbered $0, ..., n-1$).
A list of pairs (i, j) to indicate employee i and j cannot be on vacation the same week (you could get **any** list of pairs, not just a list representing a tree).

Output: The minimum number of 0 labels that are required in a labeling that has no listed pairs with the same label, or ∞ if it's not possible to label.

For example, on input
$n = 3$
$(0, 1), (1, 2), (0, 2)$
The correct output is 1

on input
$n = 4$
$(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)$
The correct output is ∞.

(a) Give a reduction that shows the problem is now NP-hard.
Remember you reduce FROM the known NP-complete problem (this one always feels backward to Robbie, be careful!)

(b) Explain why your reduction is correct. Remember to handle both directions!
We have 1-2 sentences per direction.

(c) Did we prove P=NP on homework 6? Explain why or why not. (Our explanation is 2 sentences)

5. Written: Reduction II

A **Burr Path**[1] is a path that visits at least $1/3$ of the vertices in a graph.

The **Burr Path Problem** is the following:
Input: A directed, unweighted graph on n vertices
Output:
`True` if there is a path in the graph that visits at least $\lceil n/3 \rceil$ of the vertices
`False` otherwise

For example:
If $V = \{a, b, c, d, e, f, g\}$, $E = \{(a, b), (c, d)(c, e), (f, g)\}$, then return `False` (there is no path that uses $\lceil 7/3 \rceil = 3$ different vertices)
On a 9 vertex cycle, the answer is `True`, any three consecutive vertices (and the edges between them) form a Burr Path.

Burr Path is NP-complete.

Someone suggests the following reduction from Hamiltonian Path:
"Don't change the graph at all, just take the output of the hypothetical Burr Path solver. If there is a Burr Path, there must be a Hamiltonian Path too."

The reduction is not valid.

(a) Give a graph on which the reduction fails, and explain why it is incorrect (our explanation is 2 sentences).

(b) Give a correct reduction from Hamiltonian Path to Burr Path.

(c) Explain why your reduction is correct. Remember to handle both directions! Our explanations are 2-3 sentences each.

[1] Like a Hamiltonian Path, but not as good. Hamiltonian Paths aren't named after Alexander Hamilton, but this is the best name we could think of.

6. Written: Approximation

For an undirected graph $G = (V, E)$, a **matching** is a subset E' of E such that no two edges in E' share an endpoint. Consider the following LP for solving the maximum matching problem, where each edge e in E has a corresponding decision variable x_e:

$$\begin{aligned} \text{maximize:} \quad & \sum_{e\text{: edge in } E} x_e & \\ \text{subject to:} \quad & \sum_{e\text{: } u \text{ is an endpoint of } e} x_e \leq 1 & \text{(for each vertex } u \text{ in } V\text{)} \\ & 0 \leq x_e \leq 1 & \text{(for each edge } e \text{ in } E\text{)} \end{aligned}$$

Your output might be fractional, but you can assume that every variable is set to $0, 1$, or $1/2$.[2]. We'll round to an integer matching, while ensuring that if the LP finds a (fractional) matching of value k (that is, the objective value is k), you find a (true) matching of at least $2k/3$ edges.

(a) Describe how to produce an integer matching (your algorithm should run in polynomial time, but do not worry about getting the best possible efficiency; we won't deduct for slow polynomial time algorithms)

Hint: Consider the subgraph $H = (V, F)$, where F consists of only the edges e in E for which $x_e = 1/2$. What can be said about the degree of any vertex in H?

(b) Argue that your algorithm produces a "real" matching, i.e. that you no longer have any fractional variables, and that the subset of edges you've proposed satisfies the definition of matching. (Our argument is 3 sentences, but you may need more.)

(c) Argue that your algorithm is polynomial time. (Our argument is 2 sentences, but you may need more.)

(d) Argue that your true matching has at least $2k/3$ edges, where $k = \sum_e x_e$ for the values of x_e that were set by the LP.(Our argument is 2-3 sentences, but you may need more.)

(e) Give an example of an input graph G where a maximum value of the LP really is more than the value of any matching.

Hint: your graph must not be bipartite for this to be possible.

[2]For this particular type of LP, there are some LP solvers which can do that automatically. You don't need to worry about why (look up "basic feasible solution" if you're really curious); for designing the algorithm you just need to know that this is possible

7. Written: Review Subarray DP

Recall that a **contiguous subarray** is all of the elements in an array between indices i and j, inclusive (and if $j < i$ we define it to be the empty array).

Call a subarray is **nearly contiguous** if it is contiguous (i.e. contains all elements between indices i and j for some i, j) or if it contains all but one of the elements between i and j.

For example, in the array $[0, 1, 2, 3, 4]$,

- $[0, 2, 3]$ is nearly contiguous (from 0 to 3, skipping 1), sum is 5
- $[0, 1, 2, 3]$ is nearly contiguous (because it is contiguous from 0 to 3), sum is 6.
- $[2, 4]$ is nearly contiguous (from 2 to 4, skipping 3), sum is 6.
- $[3]$ is nearly contiguous (because it is contiguous from 3 to 3), sum is 3.
- $[]$ is nearly contiguous (because it is contiguous from 1 to 0), sum is 0.
- $[0, 2, 4]$ is **not** nearly contiguous (because you'd have to remove two elements).

The sum of a nearly contiguous subarray is the sum of the included elements.

Given `int[] A`, your task is to return the maximum sum of a nearly contiguous subarray.

For example, on input $[10, 9, -3, 4, -100, -20, 15, -5, 9]$ your algorithm should return 24 (corresponding to $i = 6, j = 8$ and skipping the -5).

(a) Define one or more recurrences to solve this problem.

(b) Give English descriptions (1-2 sentences each should suffice) for what your recurrences calculate. Be sure to mention what any parameter(s) mean.

(c) How do you caclulate your final overall answer (e.g. what parameters input to which of your recurrences do you check).

(d) What memoization structure(s) would you use?

(e) What would the running time of your algorithm be (you do **not** have to write the code). Justify in 1-3 sentences.

8. Written: Board game

One of your TAs has designed a new board game for you to try. You are given a k-by-k chess board and k pieces labeled from 1 to k respectively. The game challenge you to place all the pieces on the board without violating any of the conditions, which are given in the form of two 2D integer arrays `rowConds` and `colConds`:

- `rowConds`$[i] = [a, b]$ says that the piece labeled with number a must be in a row that is strictly above the row at which the piece with label b appears.
- Similarly, `colConds`$[i] = [c, d]$ says that the piece labeled with number c must be in a column that is strictly to the left of the column at which the piece with label d appears.

Example 1:

Sample Input:

$k = 3$ (your board is 3 by 3 and your pieces are labeled 1, 2, and 3)

`rowConds` $= [[1, 2], [3, 2]]$

`colConds` $= [[2, 1], [3, 2]]$

Correct Output Board: $[[3, 0, 0], [0, 0, 1], [0, 2, 0]]$

3		
		1
	2	

Explanation: Notice that in the output, piece 3 is in row 0, piece 1 is in row 1, and piece 2 is in row 2, so this placement satisfies all the row conditions: piece 1 should be placed above piece 2 in the board and piece 3 should be placed above piece 2 in the board.

Now let's look at column conditions. In our output board piece 3 is in column 0, piece 2 is in column 1, and piece 1 is in column 2, therefore it also satisfies all the column conditions: piece 2 is placed to the left of 1, piece 3 is placed to the left of piece 2. Similarly, $[[0, 0, 1], [3, 0, 0], [0, 2, 0]]$ here is also a valid output

Example 2:

Sample Input:

$k = 4$

`rowConds` $= [[4, 3], [1, 4], [1, 2]]$

`colConds` $= [[1, 4], [2, 1], [2, 4], [3, 1], [3, 2], [3, 4]]$

Correct Output Board: $[[0, 0, 1, 0], [0, 2, 0, 0], [0, 0, 0, 4], [3, 0, 0, 0]]$

		1	
	2		
			4
3			

Example 3:

Sample Input:

$k = 3$

`rowConds` $= [[1, 2], [2, 3], [3, 1]]$

`colConds` $= [[3, 2]]$

Correct Output Board: [ ]

Explanation: There is no placement that can satisfy all of the row conditions. The first and second row conditions contradicts the third row condition. If piece 2 is below piece 1 and piece 3 is below piece 2 then piece 3 cannot be above piece 1.

(a) Describe the graph(s) you want to use for this problem. Explain what your nodes and edges are going to be for each graph.(hint: our staff solution used 2 graphs)

(b) Using the graph(s) you created, what kind of property do we need your graph(s) to have in order to have a valid placement that can satisfy every row and column condition?

(c) Design an algorithm to output one valid placement given the input. Write pseudocode (or English descriptions) for your algorithm.

(d) What is the running time of your algorithm?

9. Programming: Make A Verifier

In this problem, you'll write a verifier for 3-SAT. (Take a look at Lecture 25, Slide 6 if you need a reminder of what 3-SAT is.)

Specifically, you'll be given the following inputs:

- A boolean array `X` of length n, where `X[i]` $= x_{i+1}$ for $0 \leq i < n$

  (This array encodes a proposed setting of the variables.)

- A 2d boolean array `Y` with m rows and 3 columns, where `Y[i][j]` $= y_{i+1,\,j+1}$.
- A 2d int array `Z` with m rows and 3 columns, where $1 \leq$ `Z[i][j]` $= z_{i+1,\,j+1} \leq n$ for each element `Z[i][j]`.

  (Together, `Y` and `Z` encode the constraints.)

You'll be given some proposed setting of the Boolean variables $x_1, x_2, \ldots, x_n$, and your task is to verify whether this setting of the variables will satisfy all m of the constraints, where each constraint is of the form

$$x_{z_{i,1}} == y_{i,1} \;||\; x_{z_{i,2}} == y_{i,2} \;||\; x_{z_{i,3}} == y_{i,3}$$

for $1 \leq i \leq m$. In the constraints of the form above, each $1 \leq z_{i,j} \leq n$ determines the subscript for a boolean variable. ($1 \leq j \leq 3$)

Your task is to write an efficient verifier for the problem. In other words, you should return `True` if all constraints are satisfied with the variable setting given `False` otherwise. Note that you're only writing a verifier, not a solver. 3-SAT is NP-complete, so a solver would be very inefficient. We just want an efficient verifier (is the proposed variable setting good or not?).

In order to help you translate between the abstract problem description and the code, the following Java-style pseudocode gives an example of a concrete instance of the problem for $n = 4$ and $m = 5$:

```
final boolean T = true;
final boolean F = false;

boolean[] X = {T, F, T, T};

boolean[][] Y = {{F, T, F},
                 {F, T, T},
                 {T, T, T},
                 {T, F, T},
                 {F, T, F}};

int[][] Z = {{1, 2, 3},
             {1, 2, 3},
             {3, 1, 4},
             {4, 1, 3},
             {4, 3, 2}};

/* We want to check whether the constraint

(x_1 == F || x_2 == T || x_3 = F) &&
(x_1 == F || x_2 == T || x_3 = T) &&
(x_3 == T || x_1 == T || x_4 = T) &&
(x_4 == T || x_1 == F || x_3 = T) &&
(x_4 == F || x_3 == T || x_2 = F)

is satisfied, given that the proposed
setting of the boolean variables is:

x_1 = T, x_2 = F, x_3 = T, x_4 = T */
```