The “Every Problem Gets a Hint” Problem Set

Question 1. Given a directed graph $G = (V, E)$ with a distinguished vertex $s \in V$ (s for "start"), consider the following 2-person game on $G$ in which players alternate moves: A stone is placed on vertex $s$, after which players alternate making a single move. A move consists of moving the stone from its current location to any of its "out" neighbors; that is, if the stone is on vertex $v$, then it can be moved to any $u$ such that $(v, u) \in E$. There is one additional rule: the stone can never return to a vertex it has previously visited. A player loses if, when it becomes their turn, they have no move (all out neighbors of the vertex upon which the stone currently rests have already been visited).

For some instances $(G, s)$ of this game, Player 1 may be able to force a win (that is, always be able to win regardless of the moves that Player 2 makes), and for some instances, Player 2 might be able to force a win. It turns out that determining whether a player can force a win is $NP$-hard. This problem appears to be so hard, in fact, that it is not generally believed to be in the class $NP$!

Your task: Show that if $G$ is a DAG (directed acyclic graph) then one can determine, in polynomial time, whether one of the players has a winning strategy.

Hint: Sinks are inherently losing positions. Perhaps you can label each node as inherently winning or losing: Think "Topological Sorting"!

Worth noting: This may seem like a very abstract and specific game, but, in fact, many finite 2-player games fall into this category....

Question 2. Chapter 10, Problem 1.

Hint: Try the idea we used for vertex cover (see Property 10.3): Delete some element of some $B_i$, along with $B_i$ (and perhaps some additional $B_j$'s...?) from the problem instance. Think about how this reduction to a smaller problem can help.

Question 3. Chapter 11, Number 6

Hints:

- Note: The load on a machine now depends on whether the machine is slow or fast. A slow machine takes time $t_i$ to complete job $i$ (a fast machine would take time $t_i/2$).

- So, for a slow machine, the load is the sum of the processing times of the jobs associated with that machine; for a fast machine, the load is half of that sum.

- A good greedy algorithm for this problem: When a new job comes in, assign it to the machine which is currently has the smallest load.

- Establish the following lower bounds on the optimum makespan $T^*$:
  - $T^* \geq (\sum_j t_j)/(m + 2k)$
  - $T^* \geq t_j/2$ for every job $j$

- As in the proof from class/text, now let $i$ be the machine that achieves the makespan and think about the final job $j$ assigned to it....

Question 4. Chapter 11, Number 4

Hint: Closely mimic the pricing method approach from Section 11.4

Question 5. Chapter 10, Problem 3.

Hint: There is a Hamiltonian path from $v_1$ to $v_n$ if and only if, for some $v_i$, there is an edge from $v_i$ to $v_n$ (final edge) and a Hamiltonian path (in $G - \{v_n\}$) from $v_1$ to $v_i$. This suggests dynamic programming: if we had a table $H[S, j]$ to tell us – for each subset $S$ of $V$, where $v_1 \in S$ – whether there is a path from $v_1$ to $v_j$ using exactly the vertices of $S$, we’d be set! [Google "xkcd #399" for more information.]

Due: Noon, May 3, 2019