1 Summary of Material

1.1 P vs. NP Definitions

There are 2 classes of problems that we will examine in this course: **P** and **NP**. In today's section, we will review the difference between the two, as well as get some more practice classifying something as **P**, **NP**, and **NP**-Complete.

- **P** - the set of all yes/no problems that can be deterministically solved in polynomial time, that is runs in $O(n^k)$ steps for some positive integer k.
  - **Examples from Lecture:** Basically everything we have seen so far. Shortest paths, finding MST's, dynamic programming, max flow
- **NP** - the set of all yes/no problems where one could convince you that the answer is "yes" by giving a polynomial length certificate which can be verified in polynomial time as well.
  - **Examples from Lecture:** Compositeness, 3-SAT, Integer Linear Programming, Maximum Independent Set, Vertex Cover, Maximum Clique

Note that P is a subclass of NP because any we can verify that a P algorithm is correct by simply running it, which takes polynomial time.

1.2 Reductions

A reduction is a procedure $R(x)$ which transforms the input for problem A into an input for problem B. This transformation must maintain the integrity of the answer, that is the answer to A is yes from an input x if and only if the answer to B from $R(x)$ is yes. If we can find such a procedure $R(x)$, then we can conclude that A is at least as easy as B. This is true because we know that **if we can solve B, we can solve A** by making this reduction.

To Show that A is Easy: Reduce A to something easy.

To Show that A is Hard: Reduce something else that is hard to it.

1.3 NP-Complete

These are the hardest problems in NP, which have the property that all other problems in NP reduce to them. Our time-line of proving various problems to be NP-Complete is summarized below:

(a) **Circuit-SAT**: Cook's Theorem (informal explanation offered in lecture)

(b) **3-SAT**: Circuit-SAT reduced to 3-SAT by introducing various combinations of 3 clauses to represent x being an AND, OR, or NOT gate of y and z.

(c) **Integer Linear Programming**: 3-SAT reduced to this by replacing each literal with x or $1 - x$ and then setting the sum of each clauses to be greater than 1.

(d) **Independent Set**: 3-SAT reduced to this by constructing a strange graph where nodes represented assignments of the literals, and edges connected assignments that conflicted with each other.

(e) **Vertex Cover:** : Independent Set and Vertex Cover reduce to each other by observing that C is an independent set if and only if $V - C$ is a vertex cover.

2 Practice Problems

Exercise 1. Show that the following optimization problem is in P:

Call two paths *edge-disjoint* if they have no edges in common. Given a directed graph $G = (V, E)$ and two nodes $s, t \in V$, find the maximum number of edge-disjoint paths from s to t.

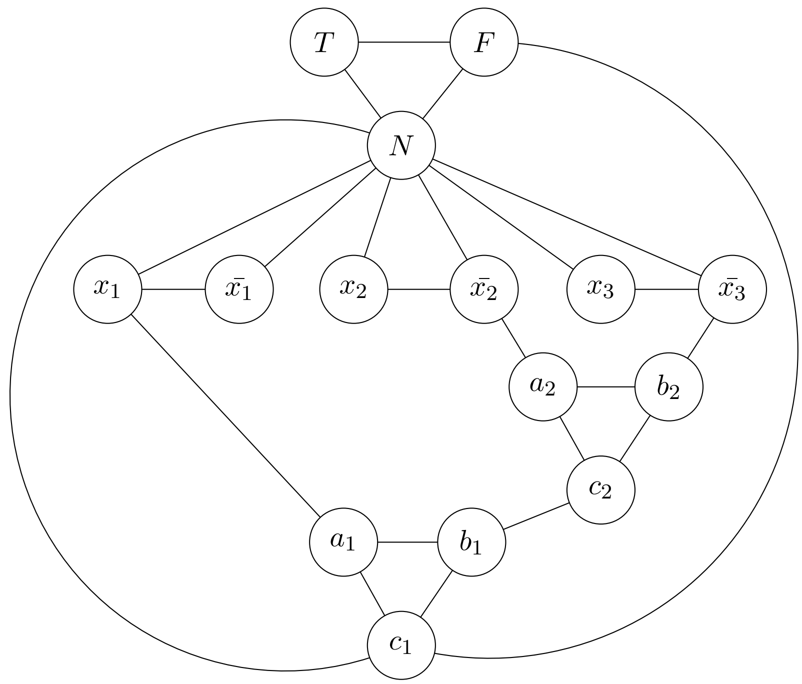
Exercise 2. Recall the *Set Cover* problem: Given a set U of elements and a collection $S_1, ..., S_m$ of subsets of U, is there a collection of at most k of these sets whose union equals U? You may remember that there is a greedy algorithm which is off by a factor of $O(\log n)$. Show that Set Cover is actually NP-Complete.

Exercise 3. A *3-coloring* of a graph $G = (V, E)$ is a assignment of colors to the vertices

$$f : V \to \{\text{red, green, blue}\}$$

such that for every edge $(u, v) \in E$, $f(u) \neq f(v)$. Show that $3-$coloring is NP-complete.

Hint: Consider the following graph with the clause $(x_1 \vee \bar{x_2} \vee \bar{x_3})$:



Exercise 4. The class NP was defined as a class of decision problems. However, typically we have an optimization problem we would like to solve and not just a decision problem. For example, in the $\textsc{VertexCover}_k$ problem we must decide whether a vertex cover exists of size at most k, as opposed to the $\textsc{MinVertexCover}$ problem of finding a vertex cover of minimum size.

(a) Show that a polynomial time algorithms for $\textsc{VertexCover}_k$ for all k imply a polynomial time algorithm for $\textsc{MinVertexCover}$.

(b) In the $\textsc{HamiltonianCycle}$ problem we must decide whether a directed graph G has a cycle of length n that touches every vertex exactly once. This is in contrast with the non-decision $\textsc{FindHamCycle}$ problem of actually *finding* a cycle. Show that a polynomial time algorithm for $\textsc{HamiltonianCycle}$ implies a polynomial time algorithm for $\textsc{FindHamCycle}$