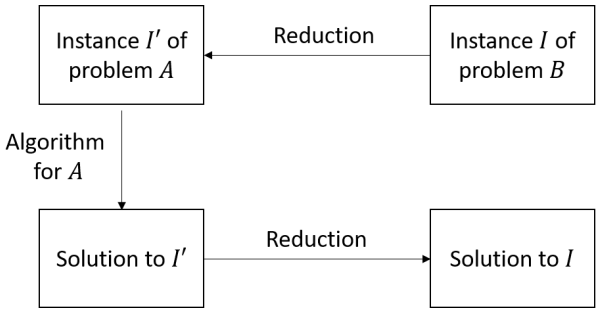
Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

Reduction: Suppose we have an algorithm to solve problem A, how can we use it to solve problem B?

This has been and will continue to be a recurring theme of the class. Examples so far include

- Use LP to solve max flow.
- Use max-flow to solve min s-t cut.
- Use minimum spanning tree to solve maximum spanning tree.

In each case, we would transform the instance I of problem B we want to solve into an instance I' of problem A that we can solve, and also describe how to take a solution for I' and transform it into a solution for I:



Importantly, the transformation should be efficient, i.e. takes polynomial time. If we can do this, we say that we have reduced problem B to problem A.

Conceptually, a efficient reduction means that if we can solve problem A efficiently, we can also solve problem B efficiently. On the other hand, if we think that B cannot be solved efficiently, we also think that A cannot be solved efficiently. Put simply, we think that A is "at least as hard" as B to solve.

To show that the reduction works, you need to prove **both**:

(1) If there is a solution for instance I' of problem A, there must be a solution to the instance I of problem B

(2) If there is a solution to instance I of B, there must be a solution to instance I' of problem A.

1 Graph Coloring Problem

An undirected graph $G = (V, E)$ is k-colorable if we can assign every vertex a color from the set $1, \ldots, k$, such that no two adjacent vertices have the same color. In the k-coloring problem, we are given a graph G and want to output "Yes" if it is k-colorable and "No" otherwise.

(a) Show how to reduce the 2-coloring problem to the 3-coloring problem. That is, describe an algorithm that takes a graph G and outputs a graph G', such that G' is 3-colorable if and only if G is 2-colorable. To prove the correctness of your algorithm, describe how to construct a 3-coloring of G' from a 2-coloring of G and vice-versa. (No runtime analysis needed).

(b) The 2-coloring problem has a $O(|V| + |E|)$-time algorithm. Does the above reduction imply an efficient algorithm for the 3-coloring problem? If yes, what is the runtime of the resulting algorithm? If no, justify your answer.

2 A Reduction Warm-up

Consider the two problems below:

> **Undirected Rudrata path problem:** we are given an undirected graph G as input, and want to determine if there exists a path in G that uses every vertex exactly once.
>
> **Longest Path in a DAG:** we are given a DAG and a variable k as input, and want to determine if there exists a path in the DAG that is of length k or more.

Now, consider the reduction below; is it correct? If so, provide a concise proof. If not, justify your answer and provide a counter-example.

> Undirected Rudrata Path can be reduced to Longest Path in a DAG. Given the undirected graph G, we will use DFS to find a traversal of G and assign directions to all the edges in G based on this traversal. In other words, the edges will point in the same direction they were traversed and back edges will be omitted, giving us a DAG. If the longest path in this DAG has $|V| - 1$ edges then there must be a Rudrata path in G, as any simple path with $|V| - 1$ edges must visit every vertex.

3 California Cycle

Prove that the following problem is NP-hard.

Input: A directed graph $G = (V, E)$ with each vertex colored blue or gold, i.e., $V = V_{\text{blue}} \cup V_{\text{gold}}$.

Goal: Determine whether there is a *Californian cycle* in G, i.e. whether there is a directed cycle through all vertices in G that alternates between blue and gold vertices.

Hint: note that the Directed Rudrata Cycle problem is NP-Complete.

If there exists a polynomial reduction from problem A to problem B, problem B is at least as hard as problem A. From this, we can define complexity class which sort of gauge 'hardness'.

Complexity Definitions

- NP: a problem in which a potential solution can be verified in polynomial time.
- P: a problem which can be solved in polynomial time.
- NP-Complete: a problem in NP which all problems in NP can reduce to.
- NP-Hard: any problem which is at least as hard as an NP-Complete problem.

Prove a problem is NP-Complete
To prove a problem is NP-Complete, you must prove the problem is in NP and it is in NP-Hard.

To prove that a problem is in NP**,** you must show there exists a polynomial verifier for it.

To prove that a problem is NP **hard,** you can reduce an NP-Complete problem to your problem.

4 NP or not NP, that is the question

For the following questions, circle the (unique) condition that would make the statement true.

(a) If B is NP-complete, then for any problem $A \in \mathsf{NP}$, there exists a polynomial-time reduction from A to B.

Always True True iff $\mathsf{P} = \mathsf{NP}$ True iff $\mathbf{P} \neq \mathbf{NP}$ Always False

(b) If B is in NP, then for any problem $A \in \mathbf{P}$, there exists a polynomial-time reduction from A to B.

Always True True iff $\mathsf{P} = \mathsf{NP}$ True iff $\mathsf{P} \neq \mathsf{NP}$ Always False

(c) 2 SAT is NP-complete.

Always True True iff $\mathsf{P} = \mathsf{NP}$ True iff $\mathsf{P} \neq \mathsf{NP}$ Always False

(d) Minimum Spanning Tree is in NP.

Always True True iff $\mathsf{P} = \mathsf{NP}$ True iff $\mathsf{P} \neq \mathsf{NP}$ Always False

5 Runtime of NP

True or False (with brief justification): Suppose we can show for some fixed k, an NP-complete problem P has a time $O(n^k)$ algorithm. Then every problem in NP has a $O(n^k)$ time algorithm.

6 Applying LP Feasibility to Disk Interiors

Consider the following problem:

Disk Interior Problem

Input: n circular disks on the two-dimensional plane specified by $\{(x_i, y_i, R_i) \mid i = 1, \ldots, n\}$ where (x_i, y_i) is the center and R_i is the radius of the i^{th} disk.

Output: A point $p^* = (x^*, y^*)$ that is inside all the disks (approximately). Formally, the point p^* must be within distance ϵ from the interior of every disk.

(a) Prove that the intersection of a family of convex sets is convex. This will imply that the intersection of a family of disks is a convex set.

(b) Consider the convex set $\mathcal{S}$ consisting of a single circular disk centered at (α, β) with radius R. Show how to implement an ϵ-separating oracle for $\mathcal{S}$.