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.

If there exists a polynomial-time 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 Class 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 polynomial-time 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.

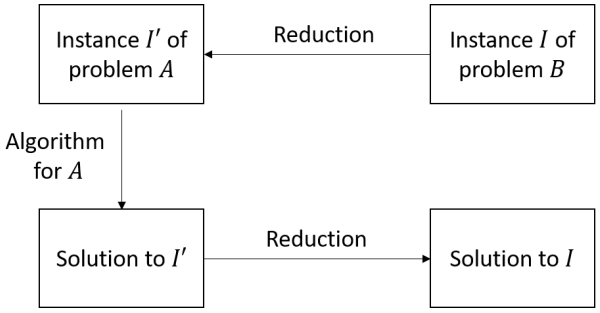
To prove that a problem is NP**-hard,** you can reduce an NP-Complete problem to your problem.

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 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.

Solution: False. The reduction f_L from an arbitary problem $L \in \mathsf{NP}$ is guaranteed to run in time $O(n^{c_L})$ and produce a problem $f(x)$ of the NP-complete problem of size $O(n^{c'_L})$ for constants c_L and c'_L. However, these can be arbitrarily larger than k.

2 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.

Solution:

(a) To construct G' from G, add a new vertex v^* to G, and connect v^* to all other vertices.

If G is 2-colorable, one can take a 2-coloring of G and assign color 3 to v^* to get a 3-coloring of G'. Hence G' is 3-colorable.

If G' is 3-colorable, since v^* is adjacent to every other vertex in the graph, in any valid 3-coloring v^* must be the only vertex of its color. This means all remaining vertices only use 2 colors total, i.e. G is 2-colorable. We can recover the 2-coloring of G from the 3-coloring of G' by just using the 3-coloring of G', ignoring v^* (and remapping the colors in the 3-coloring except for v^*'s color to colors 1 and 2).

(b) No, the reduction is in the wrong direction. This reduction shows 3-coloring is at least as hard as 2-coloring, but it could be much harder.

(Comment: Indeed, it is suspected that there is no 3-coloring algorithm running in time $O(2^{|V|^{.99}})$. This is part of a common and somewhat mystical trend we will see more examples of very soon: lots of problems go from easy to hard when we change a 2 to a 3 in the problem description.)

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.

Solution: We reduce Directed Rudrata Cycle to Californian Cycle, thus proving the NP-hardness of Californian Cycle. Given a directed graph $G = (V, E)$, we construct a new graph $G' = (V', E')$ as follows:

- V' contains two copies of the vertex set V. So, for each $v \in V$, there's a corresponding blue node v_b and a gold node v_g in V'.
- For each $v \in V$, add an edge from v_b to v_g in G'.
- For each $(u, v) \in E$, add edge (u_g, v_b) in G'.

  Another way to view this is that for each node $v \in V$, we are redirecting all its incoming nodes to v_g, and all its outgoing nodes originate from v_b (in G').

To see why this works correctly, observe that every cycle in G, will have a corresponding cycle in G' that alternates between blue and gold vertices. So if G contains a Rudrata cycle, then G' will contain a Californian cycle. Also, every edge in G' that goes from a blue node to a gold node will have both its end points correspond to the same vertex in G. So, G' has a Californian cycle, we can ignore all the blue-gold edges in that cycle and obtain a Rudrata cycle in G.

4 Cycle Cover

In the cycle cover problem, we have a directed graph G, and our goal is to find a set of directed cycles $C_1, C_2, \ldots C_k$ in G such that every vertex appears in exactly one cycle (a cycle cannot revisit vertices, e.g. $a \to b \to a \to c \to a$ is not a valid cycle, but $a \to b \to c \to a$ is), or declare none exists.

In the bipartite perfect matching problem, we have a undirected bipartite graph (a graph where the vertices can be split into L, R, and there are no edges between two vertices in L or two vertices in R), and our goal is to find a set of edges in this graph such that every vertex is adjacent to exactly one edge in the set, or declare none exists.

Give a reduction from cycle cover to bipartite perfect matching.

(Hint: In a cycle cover, every vertex has one incoming and one outgoing edge.)

Solution: Given the cycle cover instance G, we create a bipartite graph G' where L has one vertex v_L for every vertex in G, and R has one vertex v_R for every vertex in G. For an edge (u, v) in G, we add an edge (u_L, v_R) in the bipartite graph. We claim that G has a cycle cover if and only if G' has a perfect matching.

If G has a cycle cover, then the corresponding edges in the bipartite graph are a bipartite perfect matching: The cycle cover has exactly one edge entering each vertex so each v_R has exactly one edge adjacent to it, and the cycle cover has exactly one edge leaving each vertex, so each v_L has exactly one edge adjacent to it.

If G' has a perfect matching, then G has a cycle cover, which is formed by taking the edges in G corresponding to edges in G': If we have e.g. the edges $(a_L, b_R), (b_L, c_R), \ldots, (z_L, a_R)$ in the perfect matching, we include the cycle $a \to b \to c \to \ldots, z \to a$ in G in the cycle cover. Since v_L and v_R are both adjacent to some edge, every vertex will be included in the corresponding cycle cover.

