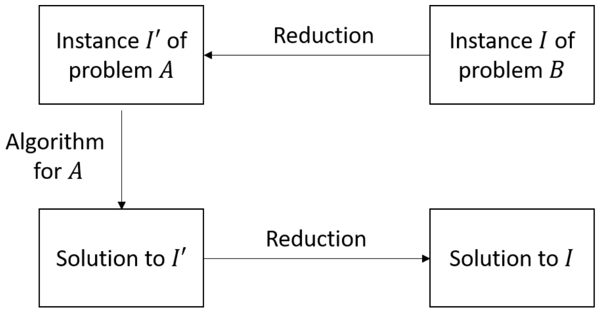
Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings,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

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.)

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.

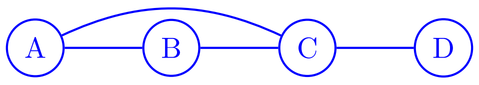
> Undirected Rudrata Path can be reduced to Longest Path in a DAG. Given the undirected graph G, we will use DFS to find a traversal of G and assign directions to all the edges in G based on this traversal. In other words, the edges will point in the same direction they were traversed and back edges 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.

Solution:

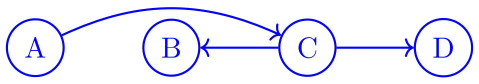
It is incorrect.

It is true that if the longest path in the DAG has length $|V|-1$ then there is a Rudrata path in G. However, to prove a reduction correct, **you have to prove both directions**. That is, if you have reduced problem A to problem B by transforming instance I to instance I' then you should prove that I has a solution **if and only if** I' has a solution. In the above "reduction," one direction does not hold. Specifically, if G has a Rudrata path then the DAG that we produce does not necessarily have a path of length $|V|-1$–it depends on how we choose directions for the edges.

For a concrete counterexample, consider the following graph:



It is possible that when traversing this graph by DFS, node C will be encountered before node B and thus the DAG produced will be



which does not have a path of length 3 even though the original graph did have a Rudrata path.

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.

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 $A \in$ NP, there exists a polynomial-time reduction from A to B.

Always True True iff P = NP True iff **P** $\neq$ **NP** Always False

Solution: Always True: this is the definition of NP-hard, and all NP-complete problems are NP-hard

(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 P = NP True iff P $\neq$ NP Always False

Solution: Always true: since we have polynomial time for our reduction, we have enough time to simply solve any instance of A during the reduction.

(c) 2 SAT is NP-complete.

Always True True iff P = NP True iff P $\neq$ NP Always False

Solution: True iff P = NP:
By definition, in order to be NP-Complete a problem must be in NP, and there must exist a polynomial reduction from every problem in NP.
If P $\neq$ NP, then there does not exist a polynomial time reduction from NP-Complete problems like 3-SAT to 2-SAT.
If P = NP, then a polynomial reduction is as follows:
since P = NP there must exist a polynomial times algorithm to solve $3 - SAT$. Thus, when we are preprocessing 3-SAT we can solve for whether there exists a solution in the instance or not. If the instance has a solution, then we will map it to an instance of $2 - SAT$ that has a solution, and if it doesn't have a solution, we will map it to an instance that doesn't have a solution. Thus all problems in NP will have a polynomial time reduction to 2-SAT as all problems in NP are reducible to 3-SAT.

(d) Minimum Spanning Tree is in NP.

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

Solution: Always True. MST is solvable in polynomial time, which means it is verifiable in polynomial time.
Note that explicitly, the decision problem would be "does there exist a spanning tree whose cost is less than a budget b?".

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.

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.

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 $\{(x_i, y_i, R_i) \mid i = 1, \dots, n\}$ 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.

Solution: Here are two proofs of the fact.

By definition, a convex set is an intersection of halfspaces. So let us say, if

$$\mathcal{S}_1 = \cap_{H \in \mathcal{H}_1} H$$

and

$$\mathcal{S}_2 = \cap_{H \in \mathcal{H}_2} H$$

then

$$\mathcal{S}_1 \cap \mathcal{S}_2 = \cap_{H \in \mathcal{H}_1} H \bigcap \cap_{H \in \mathcal{H}_2} H$$

is also an intersection of halfspaces, so is a convex set.

Alternatively, a set $\mathcal{S}$ is convex is for all $x, y \in \mathcal{S}$, the line joining $x, y \in \mathcal{S}$.

Now consider any pair of points $x, y \in \mathcal{S}_1 \cap \mathcal{S}_2$. Since $\mathcal{S}_1$ is convex, the line joining x, y is in $\mathcal{S}_1$. Similarly, the line joining x, y is in $\mathcal{S}_2$. So the line joining them is in $\mathcal{S}_1 \cap \mathcal{S}_2$. Therefore $\mathcal{S}_1 \cap \mathcal{S}_2$ is a convex set.

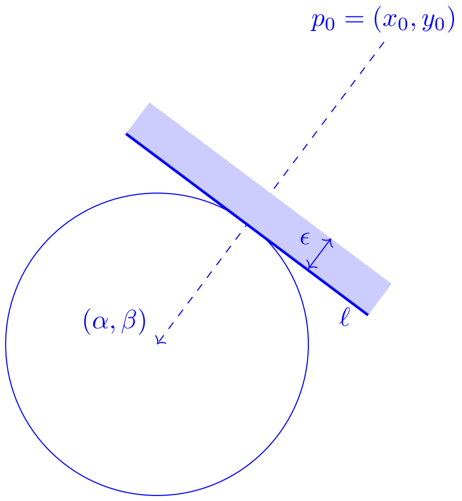
(b) Consider the convex set $\mathcal{S}$ consisting of a single circular disk centered at (α, β) with radius R. Show how to implement an ϵ-separating oracle for $\mathcal{S}$.

Solution: An ϵ-separating oracle for the circular disk is an algorithm for the following problem:

Input: Given a point $p_0 = (x_0, y_0;)$ that is at least ϵ-away from the disk, i.e., at least ϵ distance outside the circular disk.

Output: A halfspace (a.k.a., a separating line in 2-dimensions) such that circular disk is on one side of the line, but p_0 is at least ϵ-away on the other side.

The picture is shown below:



The separating line ℓ is as shown in the picture, its a tangent to the disk.

The normal to the line is the line joining $X = (x_1, x_2)$ to the center (α, β), so it is the vector:

$$w = \frac{1}{\sqrt{(\alpha - x_1)^2 + (\beta - x_2)^2}} (\alpha - x_1, \beta - x_2)$$

Here we are dividing by $\frac{1}{\sqrt{(\alpha - x_1)^2 + (\beta - x_2)^2}}$ just to make w a unit vector.

So the separating line is of the form $\langle w, X \rangle - \theta = w_1 x + w_2 x_2 - \theta = 0$ for some value of θ. Note that as we vary θ we get lines parallel to this tangent, and some fixed value of θ we have the line touching the disk.

The value of θ is not important for what we will do next, but here we will work it out just for completeness. You may skip this calculation.

The point where tangent touches the circle is,

$P = (\alpha, \beta) - R \cdot w$

Since P must be on the line, we get that θ must satisfy: $\langle w, P \rangle - \theta = 0$

So we get the equation of the line to be,

$$w_1 x_1 + w_2 x_2 - (\langle w, P \rangle) = 0$$