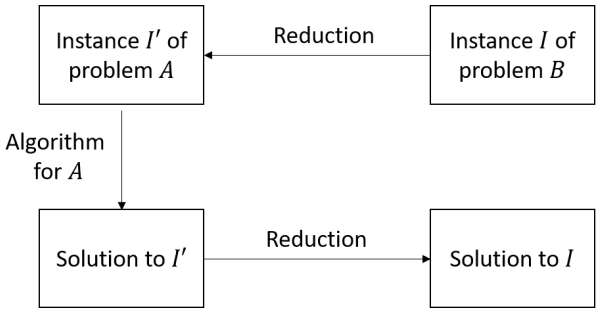
Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings,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

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 Optimization versus Search

Recall the following definition of the Traveling Salesman Problem, which we will call SEARCH-TSP. We are given a complete graph G of whose edges are weighted and a budget b. We want to find a tour (i.e., path) which passes through all the nodes of G and has length $\leq b$, if such a tour exists.

The optimization version of this problem (which we call MIN-TSP) asks directly for the shortest tour.

(a) Show that if SEARCH-TSP can be solved in polynomial time, then so can MIN-TSP.

(b) Do the reverse of (a), namely, show that if MIN-TSP can be solved in polynomial time, then so can SEARCH-TSP.

Solution:

(a) Do a binary search over all possible lengths of the optimal tour, going from 0 to the sum of all distances. Note that binary search is necessary here and we can't just increment the value of b by 1 each time since the sum of all distance is exponential in the size of the input.

(b) Run tsp-opt and return the optimal value. If it is greater than b, then no solution exists, by the definition of optimality.

2 Graph Coloring Problem

An undirected graph $G = (V, E)$ is k-colorable if we can assign every vertex a color from the set $1, \cdots, k$, 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 Set Cover vs. LP

Consider a variant of the Set Cover problem: given a collection $\{S_1, S_2, \ldots, S_m\}$ where each S_i is a subset of the universe $U = \{1, 2, \ldots, n\}$, and a budget b, we want to find a group of up to b sets from the collection such that their union contains all of U, or determine that no such group exists.

Additionally, consider an *integer linear program (ILP)*, namely a linear program where all decision variables are constrained to be non-negative integers. (we haven't talked about how to solve such a problem, but don't worry about that for now!)

(a) First, let's solve a simplified version of Set Cover where we only want to know whether a suitable group of sets exists, and don't care about the specific group of sets itself. Describe how to formulate an integer linear program that is feasible if and only if there exists a group of b sets whose union contains U.

(b) Now, if we are given a feasible (integer) point of the ILP we just constructed, describe how to find a specific group of sets that solves the set cover problem.

Solution:

(a) Our ILP will have m decision variables, where the i-th variable x_i represents whether the set S_i is included in our group. Each x_i will be constrained to be an integer between 0 and 1 (inclusive).

The objective function doesn't matter in this instance since we are only concerned with the feasibility of the ILP.

For each element of the universe $j \in U$, we will create the constraint

$$\sum_{i=1}^{m} x_i \cdot 1\{j \in S_i\} \geq 1$$

where

$$1\{j \in S_i\} = \begin{cases} 1 \text{ if } j \in S_i \\ 0 \text{ otherwise} \end{cases}$$

which ensures that we select at least one set to cover each element in U.

Additionally, we will add the constraint

$$\sum_{i=1}^{m} x_i \leq b$$

to ensure that we select at most b sets.

(b) For each x_i, include S_i in our group of sets if $x_i = 1$, otherwise do not include S_i.

4 Bad Reductions

In each part we make a wrong claim about some reduction. Explain for each one why the claim is wrong.

(a) The shortest simple path problem with non-negative edge weights can be reduced to the longest simple path problem by just negating the weights of all edges. There is an efficient algorithm for the shortest simple path problem with non-negative edge weights, so there is also an efficient algorithm for the longest path problem.

(b) We have a reduction from problem B to problem A that takes an instance of B of size n, and creates a corresponding instance of A of size n^2. There is an algorithm that solves A in quadratic time. So our reduction also gives an algorithm that solves B in quadratic time.

(c) We have a reduction from problem B to problem A that takes an instance of B of size n, and creates a corresponding instance of A of size n in $O(n^2)$ time. There is an algorithm that solves A in linear time. So our reduction also gives an algorithm that solves B in linear time.

(d) Minimum vertex cover can be reduced to shortest path in the following way: Given a graph G, if the minimum vertex cover in G has size k, we can create a new graph G' where the shortest path from s to t in G' has length k. The shortest path length in G' and size of the minimum vertex cover in G are the same, so if we have an efficient algorithm for shortest path, we also have one for vertex cover.

Solution:

(a) The reduction is in the wrong direction. By reducing shortest simple path to longest simple path, we showed that longest simple path is at least as hard as shortest simple path, but the longest simple path problem could be much harder (indeed, we don't know of any algorithm taking less than exponential time).

(b) The reduction blows up the size of the instance, so running an algorithm on the instance of A would actually take quartic/$O(n^4)$ time, not quadratic time. Furthermore, the runtime of the reduction step is polynomial in n, so it is possible for it to take $O(n^d)$ time for some $d > 2$, making the overall algorithm slower than quadratic time even before solving the instance of A.

(c) Now the reduction doesn't blow up the size of the instance, but it takes $O(n^2)$ time, so combining the reduction and algorithm for A only gives an $O(n^2)$ time algorithm.

(d) This reduction needs an algorithm for minimum vertex cover to compute what k is, which defeats the point of the reduction. Effectively, this just shows that minimum vertex cover is at least as hard as itself.