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 $I$:

![Reduction Diagram]

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

1 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.

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.

3 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. $a \rightarrow b \rightarrow a \rightarrow c \rightarrow a$ is not a valid cycle, but $a \rightarrow b \rightarrow c \rightarrow a$ 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.)

4 Decision vs. Search vs. Optimization

Recall that a vertex cover is a set of vertices in a graph such that every edge is adjacent to at least one vertex in this set.

The following are three formulations of the VERTEX COVER problem:

- As a decision problem: Given a graph $G$, return TRUE if it has a vertex cover of size at most $b$, and FALSE otherwise.
- As a search problem: Given a graph $G$, find a vertex cover of size at most $b$ (that is, return the actual vertices), or report that none exists.
- As an optimization problem: Given a graph $G$, find a minimum vertex cover.

At first glance, it may seem that search should be harder than decision, and that optimization should be even harder. We will show that if any one can be solved in polynomial time, so can the others.
(a) Suppose you are handed a black box that solves VERTEX COVER (DECISION) in polynomial time. Give an algorithm that solves VERTEX COVER (SEARCH) in polynomial time.

(b) Similarly, suppose we know how to solve VERTEX COVER (SEARCH) in polynomial time. Give an algorithm that solves VERTEX COVER (OPTIMIZATION) in polynomial time.