1 NP Completeness

1.1 Defining P and NP

There are two classes of problems that we will examine in this course: P and NP. In today’s section, we will review the difference between the two, as well as get some more practice classifying something as P, NP, and NP-Complete.

- **P (Polynomial)** – the set of all problems with yes/no answers that can be answered in polynomial time (i.e. a solution can be found in $O(n^k)$ steps for some non-negative integer $k$).
- **NP (Nondeterministic Polynomial)** – the set of all problems with yes/no answers that can be verified in polynomial time using a “certificate” (i.e. we can determine whether the answer is yes or no in $O(n^k)$ steps using the certificate).

Almost everything we’ve seen in class so far has been a P-class problem (shortest paths, MSTs, dynamic programming, max flow, etc.). Examples of NP-class problems are 3-SAT, integer linear programming, vertex cover, max clique, and more. Note that P is a subclass of NP; any correct solution we calculate in polynomial time must be verifiable in polynomial time, since it was generated by a correct polynomial-time algorithm.

The following may be a useful characterization: P is the set of problems solvable by polynomial-time deterministic algorithms, while NP is the set of problems solvable by polynomial-time non-deterministic algorithms. Non-deterministic algorithms explore all possibilities at once (e.g. exploring all children of a node at once). Think about why this characterization works!

1.2 Reductions

A reduction is a procedure $R$ which transforms an input for problem A into an input for problem B. In a reduction, we should think of problems in the following manner: given an input (which serves to “configure” the problem), is it possible to find a solution?

As an example: the 3-SAT problem takes an expression in the form of clauses and asks “is it possible to satisfy this expression?” This transformation must maintain the integrity of the answer; that is, the answer to A for an input $x$ is the same as the answer to B for the input $R(x)$. If we are able to find such a procedure $R(x)$, then we can solve A by using B. We can conclude that B is at least as “hard” to solve as A.

Typically, if we want to show that a problem A is easy (e.g. it’s a P-class problem), we reduce A to something else that we know is easy, since then A can be no harder than that problem. If we want to show that A is a hard problem to solve (e.g. it’s a NP-class problem), we usually reduce a known, difficult problem to A; then A is at least as hard as that problem.
1.3 NP-Complete

NP-complete problems are the "hardest" problems in NP; all other problems in NP reduce to them. To show that a problem $A$ is NP-complete, we reduce a known NP-complete problem to $A$. We discovered NP-complete problems sequentially in the following way:

- **Circuit-SAT**: the first NP-complete problem, introduced by Cook's Theorem (see lecture).
- **3-SAT**: Circuit-SAT reduced to this by replacing gates with (complex) boolean expressions.
- **Integer Linear Programming**: 3-SAT reduced to this by replacing each literal with $x$ or $1 - x$ and constraining the sum of each clause to be greater than 1.
- **Independent Set**: 3-SAT reduced to this by constructing a strange graph where vertices represented possible assignments of the literals, and edges connected conflicting assignments.
- **Vertex Cover**: Independent Set and Vertex Cover reduce to each other by observing that $C$ is an independent set within $V$ if and only if $V - C$ is a vertex cover.

You may freely use the fact that these problems are NP-complete in your proofs.

1.4 Reduction Practice Problems

**Exercise 1.** Show that the following optimization problem is in P:

Call two paths *edge-disjoint* if they have no edges in common. Given a directed graph $G = (V, E)$ and two nodes $s, t \in V$, find the maximum number of edge-disjoint paths from $s$ to $t$.

**Solution**

In order to show that a problem is in P, it is sufficient to reduce it to another problem in P. We reduce this problem to maximum flow by assigning every edge in $E$ a capacity of 1. Then, the maximum flow of the graph is precisely the number of edge-disjoint $s - t$ paths. If any additional edge-disjoint $s - t$ paths exists, one unit of flow across each path would lead to additional valid flows, contradicting the assumption that $f^*$ was a maximum flow.

**Exercise 2.** Recall the Set Cover problem: given a set $U$ of elements and set of $m$ subsets $S = \{S_1, S_2, \ldots, S_m\}$ with $S_i \subseteq U$ for all $i$, is there a collection of at most $k$ of these subsets whose union equals $U$? You may remember that there is a greedy algorithm which is off by a factor of $O(\log n)$. Show that Set Cover is actually NP-Complete.

**Solution**

First, it is clear that set cover is in NP because a short certificate would be the sets themselves. We can check in polynomial time whether those sets cover all the elements in the universe. We now show that Set Cover must be NP-complete by reducing Vertex Cover to it.

Given an instance of VC with $(V, E, k)$, we let our universe $U$ be the set of all edges, $E$, and construct one set $S_u$ for each vertex $u \in V$, containing all edges connected to $u$. If there exists a valid vertex cover $V' \subseteq V$, then, by definition of the VC problem, every edge in $E$ is connected to some $v \in V'$, and is therefore contained in $S_v$. On the other hand, if a valid set cover $S' \subseteq S$ exists, then the vertices corresponding to the sets in $S'$ form a vertex cover since each edge in $E$ is covered by one of the sets in $S'$. 
Exercise 3. A 3-coloring of a graph \( G = (V, E) \) is a assignment of colors to the vertices

\[ f : V \to \{ \text{red, green, blue} \} \]

such that for every edge \( (u, v) \in E \), \( f(u) \neq f(v) \). Show that 3-coloring is NP-complete.

Hint: Consider the following graph with the clause \((x_1 \lor \overline{x}_2 \lor \overline{x}_3)\):

![Graph Image]

Solution

We reduce 3-SAT to 3-coloring. First, we “name” our three colors True, False, and None and create 3 connected nodes, one for each color. Any nodes connected to the None node must be True or False. If a valid 3-coloring exists, variables corresponding to nodes with the same color as the True node will be True (similarly for the False node).

For each variable \( x_i \), add a node corresponding to \( x_i \) and a node corresponding to \( \overline{x}_i \). Connect these nodes; this forces exactly one of them to be True. Connect both nodes with None; this forces them to be either True or False. This graph now captures the set of all possible assignments.

We now need to capture clause restrictions correctly. The key is to notice that we can connect two variables \( x, y \) to \( a, b \) of a triangle made of \( a, b \) and an “output” node \( c \) to create a “detector” gate. If \( x, y \) are colored the same, then \( c \) must be colored that same color! Otherwise, \( c \) can actually be anything we want. We can use this to detect when three variables are all False: for three variables \( x, y, z \), connect \( x, y \) to a triangle with output \( c \), and connect \( z, c \) to another triangle with output \( c' \). If all three variables are all False, then \( c' \) must be False!

Finally, for each clause, do the above using the clause’s three variables as inputs, and connect the final output to both the False and None nodes. In this way, we ensure that the three variables can’t all be False, since otherwise the final output would be False and connected to the False node, violating 3-coloring!

This reduction takes an instance of 3-SAT and converts it to an instance of 3-coloring. The size of the input grows by a factor of 6 and thus the reduction can be achieved in polynomial time. It is easy to see from the reasoning above that a valid 3-SAT assignment corresponds to a valid 3-coloring.
2 Approximation Algorithms

2.1 Definitions

Some NP-complete problems are too important to ignore even though we can’t find an optimal solution in polynomial time. We can get around the slow runtime of such problems by considering at least the following:

- Small inputs – Exponential runtime is bad for big inputs, but may be fine for small inputs.
- Special cases – There may be important special cases of an NP-complete problem that can be solved in polynomial time.
- Approximation algorithms – We decide we are satisfied with near-optimal solutions to an NP-complete problem.

An approximation algorithm returns a near-optimal solution to a problem, We can enumerate the how good our approximation algorithm approximates our optimal solution. An approximation algorithm has an approximation ratio \( \rho(n) \) if for any input of size \( n \), the cost \( C \) of the solution produced by the algorithm is within a factor of \( \rho(n) \) of the cost \( C^* \) of an optimal solution. In math:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]

An approximation algorithm with approximation ratio \( \rho(n) \) is called a \( \rho(n) \)-approximation algorithm. The approximation ratio is never less than 1, and a 1-approximation algorithm produces an optimal solution.

For example, if we are maximizing a positive value, then our approximation algorithm outputs \( C \) such that \( 0 < C \leq C^* \), and our ratio \( C^*/C \) is the factor by which the optimal solution is larger than the approximate solution. In this case, \( C/C^* \leq 1 \leq C^*/C \), so we have found a \( C^*/C \)-approximation algorithm.

If we are minimizing a positive value, then our approximation algorithm outputs \( C \) such that \( 0 < C^* \leq C \), and our ratio \( C/C^* \) is the factor by which the approximate solution is larger than the optimal solution. In this case, \( C^*/C \leq 1 \leq C/C^* \), so we have found a \( C/C^* \)-approximation algorithm.

2.2 Vertex Cover Approx with LP

Turns out we can find a 2-approximation to the weighted vertex cover problem using LP. The vertex cover problem considers a graph \( G = (V, E) \) for which each of the \( n \) vertices \( v \in V \) has a weight \( w_v \geq 0 \). We seek to find a set of vertices \( S \subseteq V \) for which every edge has at least one endpoint in \( S \). We want to find \( S \) for which \( w(S) = \sum_{v \in S} w_v \) is minimized.

Suppose we could restrict our solution to the integers (an Integer Program). Then our problem reads:

Minimize \( \sum_{v \in V} w_v x_v \) subject to the constraints \( x_u + x_v \geq 1 \) for \( (u, v) \in E \) and \( x_v \in \{0, 1\} \) for \( v \in V \).

In this case, \( x_v \) acts as an indicator for \( x_v \in S \). Solving this would give us the optimal weighted vertex cover solution. Unfortunately, solving an integer program such as this one is NP-hard, so we seek to find a way to use Linear Programming to get at least an approximation for the weighted vertex cover problem.
Now we consider the problem as a linear program by removing the constraint \( x_v \in \{0,1\} \) and replacing it with \( x_v \geq 0 \). The first issue we see is that \( x_v \) can now take on fractional values – to solve this, once we get our solution to the LP, say \( x^* = (x^*_v, \ldots, x^*_n) \), we let \( S = \{ i \in V : x^*_i \geq 0.5 \} \). That is, we round \( x^*_i \geq 1/2 \) up, and \( x^*_i < 1/2 \) down.

**Exercise 4.** Show that the \( S \) obtained from this linear program via rounding is a set cover.

**Solution**
Note that for every edge \((u,v) \in E\), we have satisfied the constraint \( x^*_u + x^*_v \geq 1 \). Because \( x^*_u, x^*_v \geq 0 \), by pigeonhole principle, we have at least one of \( x^*_u \geq 1/2 \) or \( x^*_v \geq 1/2 \). Therefore we place at least one of \( u, v \) into \( S \), and so the set cover property is satisfied.

Now that we have shown that the \( S \) obtained from this linear program is indeed a set cover, we seek to analyze the minimized weight obtained by evaluating the objective function at our solution \( x^* \).

**Exercise 5.** Given that \( w_{LP}(S) \leq w(S^*) \), show that this algorithm produces a vertex cover \( S \) for which \( w(S) \) is at most twice the optimal \( w(S^*) \). That is, show that this algorithm is a 2-approximation algorithm for the Weighted Vertex Cover problem.

**Solution**
By construction, \( S \) contains the vertices with \( x^*_v \geq 1/2 \).

\[
-w_{LP}(S) = \sum_{v \in V} w_v x^*_v \\
\geq \sum_{v \in S} w_v x^*_v \\
\geq \frac{1}{2} \sum_{v \in S} w_v \\
= \frac{1}{2} w(S).
\]

Therefore we have \( w(S) \leq 2w_{LP}(S) \). Together with the given inequality, we get

\[
w(S) \leq 2w_{LP}(S) \leq 2w(S^*)
\]

showing that our algorithm is a 2-approximation.

For a detailed exposition of the LP approach to the weighted vertex cover problem, you may refer to Chapter 11.6 in Kleinberg and Tardos.