Recitation 5: NP-Completeness

1 Definitions

**P:** The set of all decision problems $D$ for which there exists a polynomial time algorithm $A$ such that $A(x) = D(x)$. We consider a polynomial time algorithm to be "efficient".

**NP:** The set of all decision problems $D$ for which there exists a polynomial time verification algorithm $V$ such that for all inputs $x$, $D(x) = 1$ if and only if there exists a polynomial-length "certificate" $y$ such that $V(x, y) = True$.

In order to compare the "hardness" of solving different problems, we use reductions! The idea is that if I have two problems $A$ and $B$, if I can show that I can solve $A$ by using a black box that solves $B$, then I can understand the difficulty of solving $A$ in terms of the difficulty of solving $B$ plus the work required to transform a solution of $B$ to the solution of $A$.

**"Karp"-Reduction:** Let $A : X \rightarrow \{0, 1\}$ and $B : Y \rightarrow \{0, 1\}$ be decision problems. $A$ is polytime reducible to $B$, or "$A$ reduces to $B$" ($A \preceq B$), if there exists a function $R : X \rightarrow Y$ that transforms inputs to $A$ into inputs to $B$ such that $A(x) = B(R(x))$. The following picture shows how this reduction leads to an algorithm for $A$ which simply builds upon the algorithm for $B$.

![Reduction Diagram]

Solving $A$ is no harder than solving $B$. In other words, if solving $B$ is "easy" (i.e. $B \in P$), then solving $A$ is easy ($A \in P$). Equivalently, if $A$ is "hard", then $B$ is "hard". Given an algorithm for $B$, we can easily construct an algorithm for $A$.

**NP-Hard:** A decision problem $D$ is NP-Hard if all problems $Q$ in NP are polynomial time reducible to it ($Q \preceq D$ for all $Q \in NP$). That is, given an efficient algorithm which solves an NP-Hard problem $D$, we can construct an efficient algorithm for any problem in NP.

**NP-Complete:** A decision problem $D$ is NP-Complete if it is in NP and is NP-Hard.

**"Cook"-Reduction:** These are more general than Karp-reductions. Given a black box that solves $B$, a Cook-reduction gives an algorithm for solving $A$ which uses at most a polynomial number of calls to $B$ as a subroutine, and requires in addition at most polynomial time to synthesize the solution for $A$. Thus, if there exists a polynomial time algorithm to solve $B$, we can transform it into a polynomial time algorithm to solve $A$. Likewise, if $A$ is "hard", then $B$ is also "hard".
2 General Proof Techniques

1. To prove that $D \in P$, give a polynomial time algorithm to solve $D$.

2. To prove that $D \in NP$, give a polynomial time verifier $V(x, y)$ for the problem. This also involves describing the “certificate” $y$ such that

(a) $y$ is polynomial-length as a function of the input $x$.
(b) For all $x$ such that $D(x) = 1$, there exists a certificate $y$ such that $V(x, y) = True$.
(c) For all $x$ such that $D(x) = 0$, there does not exist any $y$ such that $V(x, y) = True$.

3. To prove that $D$ is NP-hard, choose any known NP-hard problem $A$, and show that $A \propto D$ by giving a reduction from $A$ to $D$. We often look for a Karp-reduction, though Cook-reductions are also valid proofs.

(a) Give a polynomial time algorithm $R$ that maps an input $x$ for problem $A$ into an input for problem $D$.
(b) Show that $A(x) = B(R(x))$ by showing that

$x \in A \implies R(x) \in B$ and $R(x) \in B \implies x \in A$.

4. To prove that $D$ is NP-complete, prove that $D \in NP$ and $D$ is NP-hard.

3 Comparing Decision, Search, and Optimization Problems

So far we have restricted ourselves to decision problems. In this section, we discuss why it is often good enough to simply consider the hardness of solving the decision problem. We specifically look at the optimization, search, and decision variants of the Clique problem.

**Max-Clique($G$)**: Given graph $G = (V, E)$, find a maximum clique (i.e. a maximum subset of vertices $C \subseteq V$ that form a complete graph).

**Find-Clique($G, k$)**: Given graph $G = (V, E)$ and integer $k$, find a clique of size $k$ if it exists (i.e. a subset of vertices $C \subseteq V$ with $|C| = k$ that form a complete graph).

**Clique($G, k$)**: Given graph $G = (V, E)$ and integer $k$, does there exist a clique of size $k$ (i.e. a subset of vertices $C \subseteq V$ with $|C| = k$ that form a complete graph)?

It is clear that optimization is harder than search which is harder than decision. Therefore if the decision problem is NP-hard, then search and optimization are likewise NP-hard. We will also prove that Max-Clique is no harder than Find-Clique which is no harder than Clique. We prove this by giving a polytime reduction from Max-Clique to Find-Clique, and a polytime reduction from Find-Clique to Clique. Therefore, given a polytime algorithm for Clique, we can construct polytime algorithms for Find-Clique and Max-Clique. This shows that studying the decision problem Clique is sufficient to understand the hardness of Find-Clique and Max-Clique as well.
1. **Reduce Max-Clique to Find-Clique**

Run Find-Clique for decreasing values of $k$ beginning with $k = |V|$ until Find-Clique($G, k$) returns a clique. This will be maximum since we search in decreasing values of $k$. This requires at most $|V|$ calls to Find-Clique, which is polynomial in the input. How would we do the reduction with fewer calls to the Find-Clique oracle? (Hint: binary search)

2. **Reduce Find-Clique to Clique**

(a) If Clique($G, k$) == No, return null.

(b) For each vertex $x_1, x_2, \ldots, x_n$ in any arbitrary ordering:
   
   i. Modify the graph $G$ by removing vertex $x_i$ and all edges incident on it.
   
   ii. If Clique($G, k$) == Yes, then we know that there exists a k-clique that does not involve vertex $x_i$. Therefore we permanently remove vertex $x_i$.
   
   iii. Otherwise, if Clique($G, k$) == No, then vertex $x_i$ must be in EVERY k-clique. Thus, we add vertex $x_i$ and its edges back into the graph $G$.

(c) There will be exactly $k$ vertices left in the graph $G$ which forms the k-clique.

This requires $|V| + 1$ calls to Clique, which is polynomial in the input. Therefore this is a valid Cook-reduction.

3. **Reducing Clique to Independent Set**

Clique: Given graph $G = (V, E)$ and integer $k$, is there a set of vertices $C \subseteq V$ with $|C| = k$ that form a complete graph?

Independent Set: Given graph $G = (V, E)$ and integer $k$, is there a set of vertices $I \subseteq V$ with $|I| = k$ such that for any $u, v \in I$, $(u, v) \notin E$?

Given that Clique is NP-Complete, we prove that Independent Set is NP-complete.

1. **Show Independent Set $\in$ NP**

To prove this, we need to prove there exists a verifier $V(x, y)$. Let $x = (G, k)$ be a “yes” input. Let $y$ be $I$ that satisfies the condition.

   It takes $O(|I|)$ to check that $|I| = k$. It takes $O(|I|^2)$ to check that for every $u, v \in V'$, $(u, v) \notin E$. This checks in polynomial time that the certificate $y$ proves that $x$ is a valid input. Therefore, Independent Set is in NP.

2. **Show Independent Set $\in$ NP-hard**

We prove this by giving a Karp-reduction of Clique to Independent Set.
(a) Given an input \( x = (G, k) \) to Clique, create input \( G' \) which has the same vertices, but has edge \((u, v)\) if and only if \((u, v) \notin E\). This takes \(O(|E|)\) time, so this reduction takes polynomial time.

(b) If \( I \) is a set of vertices that form a \( k\)-Independent Set for \( G' \), then \( C = I \) is a \( k\)-Clique for \( G \) because for \( u, v \in I \), Independent Set says that \((u, v) \notin E'\), but this implies that \((u, v) \in E\) for the Clique problem due to the method of construction. This shows that there are edges between every pair of nodes in \( C \). In addition \(|C| = k\), and so \( C \) is a \( k\)-clique.

(c) If \( C \) is a set of vertices that form a \( k\)-Clique in \( G \), then \( I = C \) is a \( k\)-Independent set for \( G' \). This is because \( u, v \in C \) implies that \((u, v) \in E\) for Clique, and this implies that \((u, v) \notin E'\) for Independent Set. Since \(|I| = |C| = k\), this shows that there are \( k \) elements in the construction \( G' \) that are not adjacent to each other.

3. This proves Clique reduces to Independent Set in polynomial time, which means that Independent Set is at least as hard as Clique, so \( k\)-Independent Set is NP-hard.

5 Reducing Vertex Cover to Set Cover

**Vertex Cover:** Given graph \( G = (V, E) \) and integer \( k \), does there exist \( Y \subseteq V \) such that \(|Y| = k\) and for each \((u, v) \in E\), either \( u \in Y \) or \( v \in Y \) (or both)?

**Set Cover:** Given a set \( S \) of \( n \) elements \( \{1, 2, \ldots, n\} \) and \( m \) sets \( S_1, \ldots, S_m \) where \( S_i \subseteq S \), does there exist a set of \( k \) sets \( S_{i_1}, \ldots, S_{i_k} \) such that \( S_{i_1} \cup \cdots \cup S_{i_k} = S \)?

Given that Vertex Cover is NP-complete, we prove that Set Cover is NP-Complete.

1. **Show Set Cover \( \in \text{NP} \):** To prove this, we need to prove there exists a verifier \( \mathcal{V}(x, y) \). Let \( x = S, S_1, \ldots, S_m \) be a “yes” input. Let \( y \) be \( S_{i_1}, \ldots, S_{i_k} \) that satisfies the condition.

   In \( O(k) \) time, we can figure whether or not we have exactly \( k \) sets. In \( O(kn) \) time we can determine whether or not all elements in \( S \) is in the union. This proves we have a polynomial time verifier, which means that Set Cover is in NP.

2. **Show Set Cover \( \in \text{NP-Hard} \):** To prove this, we reduce Vertex Cover to Set Cover.

   (a) For an input \( x = (G, k) \) to Vertex Cover, we make \( R(x) = S, S_1, \ldots, S_m \). Let \( S \) be the set of all edges \( e_j \in E \). For each \( v_i \in V \), create set \( S_i \). This set contains the set of edges \( e_j \) that touch \( v_i \). This new input is polynomial because \(|S| = |E|\) and each set \( S_i \) has size at most \(|E|\) and there are \(|V|\) sets.

   (b) If there is a \( k\)-Vertex Cover, there is a \( k\)-Set Cover. If the vertex \( v_i \in V' \) is part of the vertex cover, then \( S_i \) is part of the set cover. Since every edge \( e_j \in E \) is incident to some vertex \( v_i \in V' \), this means that every element \( e_j \in E \) is covered the set \( S_i \).

   (c) If there is a \( k\)-Set Cover, there is a \( k\)-Vertex Cover. If \( S_i \) is in the set cover, choose \( v_i \) to be in the vertex cover. Every element \( e_j \) is contained in some set \( S_j \). By construction,
this means every edge \( e_j \) is incident to the vertex \( v_i \) that got chosen. Since there are \( k \) sets, there will be \( k \) vertices chosen for the vertex cover.

6 Reducing SAT to 3-SAT

The boolean operator \( \land \) denotes AND (i.e. conjunction), the operator \( \lor \) denotes OR (i.e. disjunction), and the operator \( \neg \) denotes NOT (i.e. negation). A literal \( x_1 \) is a boolean variable. A clause is a disjunction of literals (ex: \( x_1 \lor \neg x_2 \lor x_3 \)). A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses (ex: \((x_1 \lor \neg x_4) \land (\neg x_2 \lor x_3 \lor x_4)\)). The formula is satisfiable if there exists an assignment of TRUE/FALSE to each literal such that the formula evaluates to TRUE.

**SAT:** Given a CNF formula consisting of clauses \( C = \{c_1, c_2, \ldots, c_n\} \) and literals \( X = \{x_1, x_2, \ldots, x_k\} \), does there exist an assignment of TRUE/FALSE to the literals such that every clause is TRUE?

**3-SAT:** Given a CNF formula consisting of clauses \( C = \{c_1, c_2, \ldots, c_n\} \) and literals \( X = \{x_1, x_2, \ldots, x_k\} \) such that each clause involves exactly three literals, does there exist an assignment of TRUE/FALSE to the literals such that every clause is TRUE?

Given that SAT is NP-complete, we prove that 3-SAT NP-Complete.

1. **Show 3-SAT \( \in \) NP:** To prove this, we need to prove there exists a verifier \( V(f, y) \). Let the input \( f \) denote the CNF formula. Let \( y \) be the vector assigning values to each literal such that \( f(y) \) evaluates to TRUE.

   Let \( V(f, y) \) be the algorithm that evaluates formula \( f(y) \) and outputs the result \( V(f, y) = f(y) \). The time it takes to evaluate \( f(y) \) (i.e. check that each clause is true) is at most three times \( n \), or the number of clauses, since each clause has exactly 3 literals. This is a valid polynomial time verifier, which means that 3-SAT is in NP.

2. **Show 3-SAT \( \in \) NP-Hard:** To prove this, we reduce SAT to 3-SAT.

   (a) Given a CNF formula consisting of clauses \( C = \{c_1, c_2, \ldots, c_n\} \) over literals \( X = \{x_1, x_2, \ldots, x_k\} \), we construct a new collection of 3-literal clauses over a new set of literals which consists of the original variables plus sets of additional variables. For each clause \( c_i \in C \), replace it by a collection of 3-literal clauses over literals that appear in \( c_i \) plus additional literals that only appear in these constructed clauses. Let \( \{z_1, z_2, \ldots, z_m\} \subset X \) be the set of literals that appear in clause \( c_i \).

   i. If \( m = 1 \), \( c_i = z_1 \). Use two additional variables \( y_{i,1}, y_{i,2} \). The following 3-CNF is equivalent to \( c_i \):
   
   \[ (z_1 \lor y_{i,1} \lor y_{i,2}) \land (z_1 \lor \neg y_{i,1} \lor y_{i,2}) \land (z_1 \lor y_{i,1} \lor \neg y_{i,2}) \land (z_1 \lor \neg y_{i,1} \lor \neg y_{i,2}) \].

   ii. If \( m = 2 \), \( c_i = z_1 \lor z_2 \). Use one additional variable \( y_{i,1} \). The following 3-CNF is equivalent to \( c_i \):
   
   \[ (z_1 \lor z_2 \lor y_{i,1}) \land (z_1 \lor z_2 \lor \neg y_{i,1}) \].
iii. If \( m = 3 \), then \( c_i \) is a valid 3-CNF formula.

iv. If \( m > 3 \), use additional variables \( y_{i,1}, y_{i,2}, \ldots y_{i,m-3} \). The following 3-CNF is equivalent to \( c_i \):

\[
(z_1 \lor z_2 \lor y_{i,1}) \land (\neg y_{i,1} \lor z_3 \lor y_{i,2}) \land (\neg y_{i,2} \lor z_4 \lor y_{i,3}) \land \cdots \land (\neg y_{i,m-3} \lor z_m - 1 \lor z_m)
\]

(b) This is a polynomial reduction because the number of 3-literal clauses constructed in total is at most \( nk \). We at most replace each original clause with \( k \) clauses (if the original clause involved all \( k \) literals). Therefore this procedure of constructing the 3-CNF is polynomial.

(c) If there is a satisfying assignment for the variables \( X \) in the original SAT, then there is also an assignment for the 3-SAT instance that we constructed. We will use the same assignment for the variables \( X \) in 3-SAT, and we will show that there is an assignment for the new additional variables \( y_{i,j} \) such that each clause in 3-SAT is also satisfied. Consider clause \( c_i \). Again let \( \{z_1, z_2, \ldots z_m\} \subset X \) be the set of literals that appear in clause \( c_i \). If \( m \leq 3 \), the corresponding 3-literal clauses are satisfied for any choice of \( y_{i,j} \). If \( m > 3 \),

i. If \( z_1 \) or \( z_2 \) is true, assign all additional variables \( y_{i,j} \) to false. This satisfies all the clauses the correspond to \( c_i \).

ii. If \( z_{m-1} \) or \( z_m \) is true, assign all additional variables \( y_{i,j} \) to true. Again this satisfies all clauses corresponding to \( c_i \).

iii. If \( z_q \) is true, assign \( y_{i,j} \) to true for \( 1 \leq j \leq q-2 \), and assign false for \( q-1 \leq j \leq m-3 \). Again this satisfies all clauses corresponding to \( c_i \).

(d) If there is a satisfying assignment for the constructed instance to 3-SAT, we can use the same assignment restricted to the variables \( X \), and it will be a satisfying assignment for SAT.