In this lecture we will define the class NP, a class of problems that we currently do not know how to solve in polynomial time. We will define the hardest problems in this class using the notions of poly-time reducibility and NP-completeness.

8.1 P and NP

When analyzing the complexity of algorithms, it is often useful to recast the problem into a decision problem. By doing so, the problem can be thought of as a problem of verifying the membership of a given string in a language, rather than the problem of generating strings in a language. Pis the class of languages for which the membership problem can be decided in time polynomial in the size of the input string. For some languages we do not have a polynomial time algorithm for membership. However we can *verify* in polynomial time whether the given string is in the language or not, using another "witness" string. NP is the class of languages for which membership can be verified in polynomial time, given a witness string of polynomial length. More formally, $L \in$ NP iff $\exists$ a poly-time verifier V such that

$$\forall x \in L, \exists w \text{ with } |w| = \text{poly}(|x|) \text{ such that } V(x,w) \text{ accepts}$$

$$\forall x \notin L, \forall w \text{ with } |w| = \text{poly}(|x|), V(x,w) \text{ rejects}$$

The class Co-NP is the class of complements of the languages in NP, and is defined similarly: $L \in$ Co-NP iff $\exists$ a polytime verifier V such that

$$\forall x \notin L, \exists w \text{ with } |w| = \text{poly}(|x|) \text{ such that } V(x,w) \text{ accepts}$$

$$\forall x \in L, \forall w \text{ with } |w| = \text{poly}(|x|), V(x,w) \text{ rejects}$$

An example of a language in the class NP is Vertex Cover. The language contains all pairs of graphs $G = (V, E)$ with integer k for which there exists a set $S \subseteq V$ with $|S| = k$ such that $\forall e \in E$, e is incident on a vertex in S. The following is a verifier for this language: V takes as input the pair G, k and a witness set $S \subseteq V$ and verifies that all edges $e \in E$ are incident on at least one vertex in S and that $|S| \leq k$. If G has no vertex cover of size less than or equal to k then there is no witness w which can be given to the verifier which will make it accept.

8.2 P-time reducibility

The notion of reducibility allows us to convert one problem into another in polynomial time, so that if we can solve the latter, then we can also solve the former problem. The contrapositive is that if the former is an NP-hard problem, then so is the latter. When talking about decision problems, a problem A is said to reduce to problem B if there exists a polynomial time algorithm which takes as input an instance of problem A, and outputs an instance of problem B which is guaranteed to have the same result as the instance in problem A. That is, if L_A is the language of problem A, and L_B is the language of problem B, and if there is a polynomial time algorithm which maps any $l \in L_A$ into $l_B \in L_B$ and any $l' \notin L_A$ into $l'_B \notin L_B$, then problem A *polynomial-time reduces* to problem B, denoted $A \leq_P B$. This reduction is also called a Cook reduction. The practical implication of this is that if an efficient algorithm exists for problem B, then problem A can be solved by converting its instance into an instance of problem B, and applying the efficient solver to them.

8.3 NP-Completeness

Whether Pand NP are equal is one of the most prominent and important open problems in Computer Science. One tool that proves useful when considering this question is the concept of a problem being complete for a class.

Informally, the hardest problems in NP are called NP-Hard. These are problems to which every problem in NP can be poly-time reduced. Thus, a polynomial time algorithm for any one of these problems would imply that every problem in NP can be solved in polynomial time, or $\mathsf{NP} \subseteq \mathsf{P}$. We already know that $\mathsf{P} \subseteq \mathsf{NP}$ because every P-time algorithm can be thought of as an NP algorithm which takes a 0-length witness. Therefore, $\mathsf{P} = \mathsf{NP}$ iff there exists a poly-time algorithm which decides any NP-Complete problem.

Formally, $L \in$ NP-Hard iff

$$\forall L' \in \mathsf{NP}, L' \leq_P L$$

If a problem is in NP-Hard and in NP, then it is called NP-Complete.

It is worth noting that for proving NP-Completeness, typically a more restrictive form of reduction is used, namely a Karp reduction. It follows the same basic structure, but requires that only one query be made to the solution for the problem being reduced to, and that the solution be directly computable from the result of the query.

8.4 Cook-Levin Theorem

The first problem proved to be in NP-Complete was Boolean SAT, which asks, given a Boolean expression, is there a setting of variables which allows the entire expression to evaluate to True? The NP-completeness of SAT was proved independently by Cook and Levin, and is called the Cook-Levin theorem. The Cook-Levin Theorem shows that SAT is NP-Complete, by showing that a reduction exists to SAT for any problem in NP. Before proving the theorem, we give a formal definition of the SAT problem (Satisfiability Problem):

Given a boolean formula ϕ in CNF (Conjuctive Normal Form), is the formula satisfiable? In other words, we are given some boolean formula ϕ with n boolean variables $x_1, x_2, \ldots, x_n$, $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$, where each of the C_i is a clause of the form $(l_{i1} \vee l_{i2} \vee \ldots \vee l_{il})$, with each l_{ij} is a literal drawn from the set $\{x_1, x_2, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$. We need to decide whether or not there exists some setting of the variables $x_1, x_2, \ldots, x_n$ that causes the formula ϕ to be satisfied (take on a boolean value of true).

Theorem 8.4.1 *Cook-Levin Theorem: SAT is* NP*-complete.*

Proof: Suppose L is a NPproblem; then L has a polynomial time verifier V:

1. If $x \in L$, $\exists$ witness y, $V(x,y) = 1$
2. If $x \notin L$, $\forall$ witness y, $V(x,y) = 0$

We can build a circuit with polynomial size for the verifier V, since the verifier runs in polynomial time (note that this fact is nontrivial; however, it is left to the reader to verify that it is true). The circuit contains AND, OR and NOT gates. The circuit has $|x| + |y|$ sources, where $|x|$ of them are hardcoded to the values of the bits in x and the rest $|y|$ are variables.

Now to solve problem L, we only need to find a setting of $|y|$ variables in the input which causes the circuit to output a 1. That means the problem L has been reduced to determining whether or not the circuit can be caused to output a 1. The following shows how the circuit satisfication problem can be reduced to an instance of SAT.

Each gate in the circuit can be represented by a 3CNF (each clause has exactly three terms). For example:

1. The functionality of an OR gate with input a, b and output Z_i is represented as: $(a \vee b \vee \bar{Z}_i) \wedge (Z_i \vee \bar{a}) \wedge (Z_i \vee \bar{b})$.
2. The functionality of a NOT gate with input a and output Z_i is represented as: $(a \vee Z_i) \wedge (\bar{a} \vee \bar{Z}_i)$.

Notice that although some of the clauses have fewer than 3 terms, it is quite easy to pad them with independant literals to form clauses in 3CNF. The value of an independent literal won't affect the overall boolean value of the clauses.

Suppose we have q gates in V marked as $Z_1, Z_2, \ldots, Z_q$ with Z_q representing the final output of V. Each of them either takes some of the sources or some intermediate output Z_i as input. Therefore the whole circuit can be represented as a formula in CNF:

$\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_q \wedge Z_q$

where

$C_i = (t_1 \vee t_2 \vee t_3), \;\; t_1, t_2, t_3 \in (x, y, Z_1, Z_2, \ldots\, Z_q, \bar{Z}_1, \bar{Z}_2, \ldots, \bar{Z}_q)$

As we said before, even if the last clause of ϕ has only one term Z_q, we may extend ϕ to an equivalent formula in 3CNF by adding independent variables. Thus, we have shown that the circuit can be reduced to ϕ, a formula in 3CNF which is satisfiable if and only if the original circuit could be made to output a value of 1. Hence, $L \leqslant_p$ SAT. Since SAT can be easily shown to be in NP (any satisfying assignmant may be used as a certificate of membership), we may conclude that SAT is NP-Complete. ∎

Furthermore, it should be noted that throughout the proof, we restricted ourselves to formulas that were in 3CNF. In general, if we restrict the formulas under consideration to be in k-CNF for some $k \in \mathbb{Z}$, we get a subproblem of SAT which we may refer to as k-SAT. Since the above proof ensured that ϕ was in 3CNF, it actually demonstrated not only that SAT is NP-Complete, but that 3-SAT is NP-Complete as well. In fact, one may prove in general that k-SAT is NP-Complete for $k \geq 3$; for $k = 2$, however, this problem is known to be in P.

8.5 Vertex Cover

One example of an NP-Complete problem is the vertex cover problem. Given a graph, it essentially asks for the smallest subset of the vertices which cover the edges of the graph. It can be phrased both as a search problem and as a decision problem. Formally stated, the search version of the vertex cover problem is as follows:

Given: A graph $G = (V, E)$

Goal: A subset $V' \subseteq V$ such that for all $(u, v) \in E$, either $u \in V'$ or $v \in V'$, where $|V'|$ is minimized.

For a concrete example of this problem, consider the following graph:

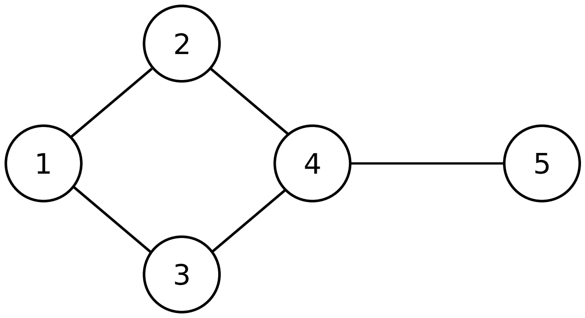

Figure 8.5.1: The nodes 1 and 4 form a minimal vertext cover

The decision variant of this problem makes only two changes to its specification: a positive integer k is added to the givens; and rather than searching for a V' of minimal size, the goal is to decide whether a V' exists with $|V'| = k$.

Furthermore, we can reduce the search version of this problem to the decision version. We know that the size of a minimal vertex cover must be between 0 and V, so if we are given access to a solution for the decision version we can just do a binary search over this range to find the exact value for the minimum possible size for a vertex cover. Once we know this, we can just check vertices one by one to see if they're in a minimal vertex cover. Let k be the size we found for a minimal vertex cover. For each vertex, remove it (and its incident edges) from the graph, and check whether it is possible to find a vertex cover of size $k - 1$ in the resultant graph. If so, the removed vertex is in a minimal cover, so reduce k by 1 and continue the process on the resulting graph. If not, add the vertex back into the graph, and try the next one. When k reaches 0, we have our vertex cover. This property of being able to solve a search problem via queries to an oracle for its decision variant is called self-reducibility.

Theorem 8.5.1 *The decision version of Vertex Cover is* NP*-Complete.*

Proof: It is immediate that Vertex Cover $\in$ NP. We can verify a proposed vertex cover of the appropriate size in polynomial time, and so can use these as certificates of membership in the language.

All we need to show then is that VC is NP-Hard. We do so by reducing 3-SAT to it. Given some

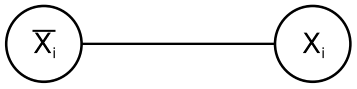

Figure 8.5.2: The gadget X_i

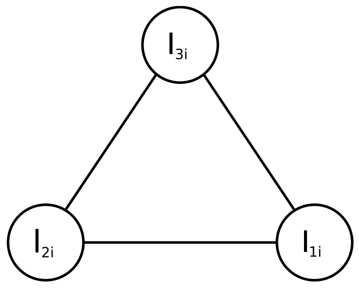

Figure 8.5.3: The gadget C_i

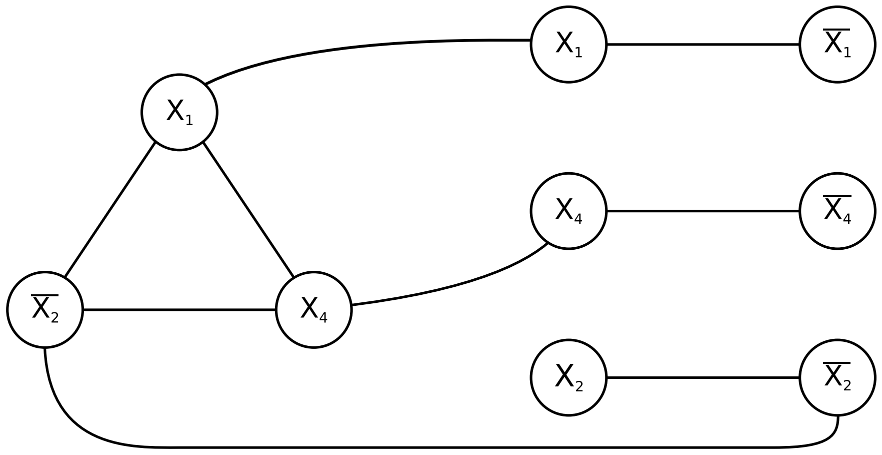

Figure 8.5.4: Connecting the gadgets

formula ϕ in 3-CNF, with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$, we will build a graph, starting with the following gadgets.

For each variable x_i, we will build a gadget X_i. X_i will have two vertices, one corresponding to each of x_i and $\bar{x}_i$, and these two vertices will be connected by an edge (see figure 8.5.2). Notice that by construction, any vertex cover must pick one of these vertices because of the edge between them.

For each clause c_i, we will build a gadget C_i. Since ϕ is in 3-CNF, we may assume that $c_i = (l_{i1} \vee l_{i2} \vee l_{i3})$ for some set of literals l_{i1}, l_{i2}, l_{i3}. The gadget C_i will consist of a complete graph of size 3, with a vertex corresponding to each of the literals in the clause c_i (see figure 8.5.3). Notice that since all of the vertices are connected by edges, any vertex cover must include at least two of them.

Given the preceding gadgets for each of the variables x_i and each of the clauses c_i, we describe how the gadgets are connected. Consider the gadget C_i, representing the clause $c_i = (l_{1i} \vee l_{2i} \vee l_{3i})$. For each of the literals in c_i, we add an edge from the corresponding vertex to the vertex for the appropriate boolean value of the literal. So if $l_{i1} = x_k$, we will add an edge from the vertex for

l_{i1} in the clause gadget C_i to the vertex for x_k in the variable gadget X_k; similarly, if $l_{i1} = \bar{x_k}$, we add and edge to the vertex for $\bar{x_k}$ in X_k. Figure 8.5.4 gives the connections for an example where $c_i = (x_1 \vee \bar{x_2} \vee x_4)$.

Now that we have our graph, we consider what we can say about its minimum vertex cover. As previously noted, at least one vertex from any variable gadget must be included in a vertex cover; similarly, at least 2 vertices from any clause gadget must be included. Since we have n variables and m clauses, we can thus see that no vertex cover can have size strictly less than $n + 2m$. Furthermore, we show that a vertex cover of exactly this size exists if and only if the original formula ϕ is satisfiable.

Assume that such a vertex cover exists. Then, for each of the gadgets X_i, it contains exactly one vertex. If we set each x_i so that the node in the vertex cover corresponds to a boolean value of true, we will have a satisfying assignment. To see this, consider any clause c_i. Since the vertex cover is of size exactly $n + 2m$, only two of the vertices in the gadget C_i are included. Now, the vertex which wasn't selected is connected to the node corresponding to its associated literal. Thus, since we have a vertex cover, that node must be in it. So, the unselected literal must have a boolean value of true in our setting of the x_i, and so the clause is satisfied.

Similarly, if an assignment of the x_i exists which satisfies ϕ, we can produce a vertex cover of size $n+2m$. In each gadget X_i, we simply choose the vertex corresponding to a boolean value of true in our satisying assignment. Then, since it is a satisfying assignment, each clause contains at least one literal with a boolean value of true. If we leave the corresponding vertex, and include the others, we get a vertex cover. The edges inside the gadget C_i are satisfied, since we have chosen all but one of its vertices; the edges from it to variable gadgets are satisfied, since for each one, either the vertex in C_i has been included or the vertex in the corresponding X_i has been chosen.

Thus, we can see that the original formula is satisfiable if and only if the constructed graph has a vertex cover of size $n + 2m$. Thus, since 3SAT is NP-Hard, we get that VC is NP-Hard as well. As we have previously noted, VC $\in$ NP, and hence VC is NP-Complete. ■