1 NP-complete problems

In the previous lecture notes, we defined the notion of completeness for a complexity class. A priori, it is not clear that complete problems even *exist* for natural complexity classes such as NP or PSPACE. Fascinatingly, completeness turns out to be a pervasive phenomenon - most natural problems in NP are either in P or NP-complete. Garey and Johnson have written an entire book, "Computers and Intractability: A Guide to the Theory of NP-completeness", which is essentially a compendium of NP-complete problems along with proofs that completeness holds. Proving that a problem is NP-complete might not seem to have direct relevance to solving that problem in practice. But in fact, such proofs give us a lot of important information. They give evidence that the problem is hard to solve, and motivate the exploration of more relaxed notions of solvability, since polynomial-time solvability in the worst case would imply the unlikely conclusion that $\mathsf{NP} = \mathsf{P}$. Also, they *connect* the complexity of the problem to that of various other problems - any solution procedure for the problem can be used to solve arbitrary problems in NP. Such connections between problems are especially striking when the problems come from different domains. It is notable that the range of problems that are NP-complete includes problems from logic (eg., Satisfiability), graph theory (eg., Clique, Vertex Cover, Colourability), optimization (Integer Linear Programming, Scheduling), number theory (Subset Sum) etc. For several sub-fields of computer science, such as databases, architecture, artificial intelligence and machine learning, natural computational problems in these fields turn out to be NP-complete. This is the case even for problems from other fields, such as statistical physics, biology and economics. It is quite extraordinary that the NP vs P problem is relevant to so many different areas of science - this is a major reason why this is considered a fundamental problem.

Ideally, we would like to show NP-completeness for *natural* problems, i.e., problems which are studied independently of their status in complexity theory. However, it is already interesting to show that there *exist* problems which are NP-complete. The most simple such proof demonstrates completeness of a somewhat unnatural problem concerning halting of non-deterministic Turing machines.

Definition 1 *The Bounded Halting problem for non-deterministic Turing machines (BHNTM) is defined as follows. An instance $< M, x, 1^t >$ belongs to $BHNTM$, where M is an encoding of an NTM, x is a binary string, and t is a number, iff M accepts x within t steps.*

In general, we will abuse notation by using M both to refer to a machine and its *encoding*. It will be clear from the context which of these is meant.

Theorem 2 *$BHNTM$ is NP-complete.*

Proof. First, we show that $BHNTM$ belongs to NP. Here, we assume the existence of an efficient universal non-deterministic Turing machine U which, given the encoding of an NTM M and an input x, outputs $M(x)$ in time which is a fixed polynomial of the time taken by M on x. Such a machine can be constructed in a similar way to deterministic efficient universal machines.

Given that U exists, the proof that $BHNTM$ is in NP is easy. To solve $BHNTM$, simply simulate U on $< M, x >$ for $p(t)$ steps, where p is a fixed polynomial such that U takes time $p(t)$ to simulate t steps of M on x. Accept if U accepts and reject otherwise. Clearly, this defines a non-deterministic Turing machine operating in polynomial time, since the machine halts within $p(t)$ steps for a fixed polynomial p, and the length of the input is at least t. The machine accepts $< M, x, 1^t$ iff U accepts $< M, x >$ in at most $p(t)$ steps, which happens iff M accepts x in at most t steps.

To prove that $BHNTM$ is NP-hard, we need to define, for each $L \in NP$, a polynomial-time computable function f such that for each x, $x \in L$ iff $f(x) \in BHNTM$. Since $L \in$ NP, there is some non-deterministic Turing machine M which decides L within $q(n)$ steps, where q is a polynomial. Now we define f as follows: for each x, $f(x) =< M, x, 1^{q(|x|)}$. Clearly, $x \in L$ iff M accepts x within $q(|x|)$ steps, which is the case iff $f(x) \in BHNTM$. We also need to show that f is polynomial-time computable. It is easy to define a polynomial-time transducer computing f, since the encoding of the machine M is fixed independent of x, and the value $q(|x|)$ can be easily be computed in unary in polynomial time for a fixed polynomial q. All the transducer needs to do is encode M, x and $1^{q(|x|)}$ together into one string - such an encoding can also be done easily in polynomial time. $\square$

One of the fundamental theorems in complexity theory is that SAT, the satisfiability problem for Boolean formulae in conjunctive normal form, is NP-complete. This result was proved by Stephen Cook in 1971, and independently by Leonid Levin. The advantage of this result over Theorem 2 is that SAT is a natural problem. This completeness result opens up the possibility of showing that several other natural problems in NP are also NP-complete by constructing reductions from SAT.

Theorem 3 *SAT is NP-complete.*

Proof. To see that $SAT \in$ NP, consider a non-deterministic machine which guesses an assignment $\vec{x}$ to the input formula ϕ, and then verifies that $\vec{x}$ satisfies ϕ. At most $|\phi|$ bits are guessed in the first phase, since a formula ϕ can have at most $|\phi|$ variables. The verification phase can also be done in polynomial time, since it simply involves evaluating a CNF formula on a given assignment. Thus the machine can be implemented to work in polynomial time.

The much harder part of the proof is to show that SAT is NP-hard. Let $L \in$ NP be an arbitrary language, and let M be a non-deterministic Turing machine accepting L in time at most $p(n)$, where p is a polynomial. Given any instance x of L with $|x| = n$, we show how to construct in polynomial time a formula ϕ_x such that M accepts x iff ϕ_x is satisfiable.

Without loss of generality, we assume $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ is a 1-tape Turing machine - this will make the reduction simpler. Note that the restriction to 1-tape Turing machines does not affect the class NP, hence we are justified in making this restriction. When we talk about 1-tape TMs here, we permit symbols to be written on the tape, unlike in the case of general multi-tape TMs, where the tape is read-only. We will also assume without loss of generality that when the machine reaches an accepting configuration, it stays there. This assumption allows us to restrict attention to computations of length exactly $p(n)$.

The central notion in the proof is that of a *computation tableau.* The computation tableau is a 2-dimensional table which represents the evolution of a computation. The rows of the tableau correspond to time steps, and the columns correspond to positions on the tape of the TM. The i'th row of the tableau represents the configuration of the TM after the i'th time step. Configurations of a 1-tape TM can be represented simply as strings over the alphabet $\Delta = Q \cup \Gamma \cup \{\#\}$, where the # symbol functions as an end-marker. The string $\#\alpha q\beta\#$, where $\alpha, \beta \in \Gamma^*$ and $q \in Q$, represents the configuration where the string α is written on the first $|\alpha|$ cells of the TM's tape, the string β followed by blanks on the rest of the tape, the finite control is in state q, and the tape head is reading the $|\alpha + 1|$'th symbol on the input tape.

Let C_i be the string representing the configuration of the TM after i steps. A matrix with $p(n)+1$ rows and $p(n)+4$ columns is a correct tableau for M on x if the rows $C_0, C_1 \ldots C_{p(n)}$ represent a correct computation of M on x. The question of whether M accepts on x now translates to the question of whether there is a correct tableau of M on x such that $C_{p(n)}$ represents an accepting configuration.

We will construct a formula ϕ_x such that assignments to the variables of ϕ_x correspond to tableaux, and an assignment satisfies ϕ_x iff the tableau corresponding to the assignment is a correct accepting tableau of M on x. The formula ϕ_x is over variables $X_{i,j,k}$, where $0 \leqslant i \leqslant p(n)$, $0 \leqslant j \leqslant p(n) + 3$, and $k \in \Delta$. Thus the number of variables is $O(p(n)^2)$. The intended interpretation of the variable $X_{i,j,k}$ being set to true is that the (i, j)'th cell of the tableau contains the symbol k.

The clauses of the formula are constraints ensuring that the intended correspondence between correct accepting tableaux and satisfying assignments to ϕ_x holds. The CNF formula ϕ_x is the conjunction of four sub-formulas in CNF - ϕ_{init}, ϕ_{accept}, ϕ_{config} and ϕ_{comp}. ϕ_{init} encodes the constraint that C_0 represents the initial configuration of M on x. ϕ_{accept} encodes the constraint that $C_{p(n)}$ represents an accepting confgiruation of M on x. ϕ_{config} encodes the constraint that each cell in the tableau holds exactly one symbol, i.e., for each i and j, there is a unique k such that $X_{i,j,k}$ is true. ϕ_{comp} encodes the constraint that for each $i, 0 \leqslant i \leqslant p(n)-1$, the $i+1$'th row of the tableau represents a configuration that can arise in one step from the configuration represented by the i'th row.

We need to describe more explicitly how to write the sub-formulas in CNF. ϕ_{init} is a CNF which specifies that the jth cell in the 0'th row of the tableau contains x_j, where x_j is the j'th symbol of the input x, when $1 \leqslant j \leqslant n$, contains the symbol # when $j = 0$ or $j = p(n)+3$, and contains the blank symbol for all other j. This condition can be written as a single conjunction of literals. ϕ_{final}

is simply a disjunction of $X_{p(n),j,q_f}$ over all $0 \leqslant j \leqslant p(n)+3$, which is clearly a CNF (with one clause!).

ϕ_{config} encodes the condition that for each i and j, at least one $X_{i,j,k}$ is true, and also at most one $X_{i,j,k}$ is true. For a given i and j, this condition can be written as a CNF where the first clause is a disjunction over all $k \in \Delta$ of $X_{i,j,k}$, and there are $\binom{k}{2}$ other clauses which say that for each pair of distinct $k_1, k_2 \in \Delta$, either X_{i,j,k_1} is false or X_{i,j,k_2} is false. ϕ_{config} is the conjunction over all i and j of the CNFs expressing the condition that the (i,j)'th cell of the tableau contains a unique symbol.

The most important sub-formula is ϕ_{comp}, which is the only part of ϕ_x which actually depends on the transition relation of the NTM M. The crucial idea when defining ϕ_{comp} is the *locality of computation.* During one step of a Turing machine computation, the configuration can only change by a bounded amount - the current tape symbol and state can change, and the tape head position can change by at most one. This is crucial to encoding the "compatibility" of consecutive rows C_i and C_{i+1} of a tableau in CNF. More precisely, we will look at fixed 2×3 "windows" of the tableau. The window corresponding to a cell (i,j) is the set of 6 cells $(i-1,j-1),(i-1,j),(i-1,j+1),(i,j-1),(i,j)$ and $(i,j+1)$. Assuming that ϕ_{init} is satisfied, whether a tableau is correct in the sense that it encodes a correct computation of the NTM M on x reduces to the question of whether all (i,j)-windows are *valid* in the sense that they occur as part of *some* computation of M. Clearly, if a tableau is correct, all (i,j)-windows will be valid. To see the other direction, assume the tableau is incorrect (in that it does not correspond to any computation of M), and let i be the first row and j the first column in that row such that the first $i-1$ rows of the tableau together with the cells up to column j in the i'th row are not consistent with any correct tableau. Since ϕ_{init} is satisfied, we have that $i \geqslant 1$. Then, by the locality of computation, the $(i,j-1)$-window will be invalid.

Now the key observation is that checking whether a window is valid can be done using a CNF of constant size. This is because the question of whether a window is valid depends on a constant number of variables - there are a constant number of cells in a window, and each one of these has a constant number of variables of ϕ_x associated with it. Now any Boolean function on a constant number of variables has a CNF of constant size. This follows from the fact that any Boolean function on t variables has a CNF of size at most $t2^t$. Let $C_{i,j}$ be the CNF checking that the (i,j)-window is valid. ϕ_{comp} is the conjunction of $C_{i,j}$ for all $1 \leqslant i \leqslant p(n)+1$, $1 \leqslant j \leqslant p(n)+2$. Clearly, the size of ϕ_{comp} is $O(p(n)^2)$.

The size of ϕ_x is polynomial in $|x|$ since both ϕ_{init} and ϕ_{accept} are of size $O(p(n))$, and ϕ_{config} and ϕ_{comp} are of size $O(p(n)^2)$. Moreover, ϕ_x can be generated from x in polynomial time. To see this, we consider in turn the complexity of generating the sub-formulae $\phi_{init}, \phi_{accept}, \phi_{config}, \phi_{comp}$. The formulae ϕ_{init}, ϕ_{accept} and ϕ_{config} all have a very simple form and it is easy to see that they can be generated efficiently. As for ϕ_{comp}, the constant size formula $C_{i,j}$ for any fixed i and j can be generated in constant time, and hence the conjunction of these formulae can be generated in time $O(p(n)^2)$.

We need to argue that M accepts x iff ϕ_x is satisfiable. If M accepts x, there is a correct accepting tableau of M on x, and by setting $X_{i,j,k}$ to be true iff the (i,j)'th cell of this tableau contains the symbol k, we derive a satisfying assignment to ϕ_x. Conversely, if ϕ_x is satisfiable, so is ϕ_{config}, and so there is some fixed tableau corresponding to any given satisfying assignment. The satisfiability of ϕ_{init} ensures that the first row of this tableau is correct. The satisfiability of ϕ_{comp} ensures that all remaining rows are correct. The satisfiability of ϕ_{accept} ensures that the final row is accepting, and hence that the tableau is a correct accepting tableau of M on x, which implies M accepts on x.

□

2 Reductions from SAT

Theorem 3 facilitates showing that several other natural problems in NP are NP-complete, since SAT is a natural problem from which to reduce. We give two simple examples here - the $3-SAT$ problem, which is a restriction of SAT, and the Integer Linear Programming problem, which is a natural optimization problem. In the Computability and Intractability course (which is a 3rd year course), several other examples of this kind are discussed.

The $3-SAT$ problem is the satisfiability problem for 3-CNF formulas, i.e., CNF formulas where every clause has at most 3 literals.

Theorem 4 *$3-SAT$ is NP-complete.*

Proof. $3-SAT$ is clearly in NP, since an assignment to a 3-CNF formula can be guessed and verified in polynomial time.

We define a polynomial-time reduction from SAT to $3-SAT$. Let ϕ be an instance of the SAT problem with clauses $C_1 \ldots C_m$. We show how to define a 3-CNF formula ϕ' such that ϕ' is satisfiable iff ϕ is satisfiable.

The reduction works clause by clause - for each clause C_i in ϕ, we define a 3-CNF formula ϕ_i over a larger variable set such that the an assignment satisfies C_i iff there is an extension of it (i.e., the assignment together with assignments to variables in ϕ_i but not in C_i) which satisfies ϕ_i. Assume wlog that $C_i = y_1 \vee y_2 \vee \ldots y_r$, where each y_i is a literal. ϕ_i is defined over the variables mentioned in C_i together with $r-2$ new variables $z_1 \ldots z_2 \ldots z_{r-2}$. These new variables are chosen afresh for each i.

The formula ϕ_i is defined as $(y_1 \vee y_2 \vee z_1) \wedge (NOT(z_1) \vee y_3 \vee z_2) \ldots (NOT(z_{r-2}) \vee y_r)$. Now notice that for C_i to be satisfied, at least one of the y_j must be true. Say y_s is true. Then by setting z_j to be true for $j \leqslant s-2$ and false for $j > s-2$, ϕ_i is satisfied as well. Conversely, if ϕ_i is satisfied, then it can't be the case that all the y_j are false, since the restriction of ϕ_i obtained by setting all y_j to false is $z_1 \wedge (NOT(z_1) \vee z_2) \wedge \ldots (NOT(z_{r-2}))$ which is unsatisfiable.

Now, we define ϕ' to be the conjunction over all i of ϕ_i. If ϕ is satisfiable, there is an extension of the satsifying assignment of ϕ which satisfies ϕ', by the argument in the previous para. Conversely, if there is a satisfying assignment to ϕ', then the projection of that satisfying assignment to the variables of ϕ satisfies ϕ. It is easy to see that ϕ' can be constructed in polynomial time from ϕ, and clearly ϕ' is a 3-CNF. $\square$

An input to the Integer Linear Programming (ILP) problem is a set of inequalities with rational co-efficients. The question is whether there is an integer assignment to the variables which satisfies all the constraints. Unlike with most NP-complete problems, it is not that easy to see that $ILP \in \mathsf{NP}$, since a witness need not necessarily have polynomial size. This is the case though, but the reasons are beyond the scope of the course.

We focus on showing NP-hardness.

Theorem 5 *ILP is NP-hard.*

Proof.

Given a CNF formula ϕ with clauses $C_1 \dots C_m$, we show how to construct a set of inequalities with rational co-efficients which is sastifiable by integer assignments to the variables iff ϕ has a satisfying assignment. Again, the reduction works clause by clause.

Consider any specific clause C_i and assume wlog that $C_i = y_1 \vee y_2 \dots y_r$, where each y_i is a literal. For each j, let x_j be the variable corresponding to the literal y_j and let z_j be a Boolean value which is 1 if $y_j = x_j$ and 0 if $y_j = NOT(x_j)$. We create an inequality corresponding to the clause C_i. The variable set over which the ILP instance is defined is the same as the variable set of ϕ, and we will assume that the names of the variables are the same as well. The inequality corresponding to C_i simply states that the sum over all $j, 1 \leqslant j \leqslant r$ of $(2z_j - 1)x_j + (1 - z_j)$ is at least 1. Note that $(2z_j - 1)x_j + (1 - z_j)$ is $(1 - x_j)$ when $y_j = NOT(x_j)$ and x_j when $y_j = x_j$.

We also add "variable constraints": inequalities for each x_j stating that $0 \leqslant x_j \leqslant 1$.

Now, if there is a satisfying assignment to ϕ, then interpreting a true assignment to a variable as 1 and a false assignment as 0 satisfies the ILP instance. Conversely, if there is an integer assignment to the ILP instance which sastifies all constraints, then each variable is either zero or one in the assignment, by the variable constraints. Now, we can interpret a variable assignment to the ILP as a truth assignment to the variables of ϕ in the natural way, and the satisfaction of the i'th constraint in the ILP instance ensures that C_i is satisfied, since at least one y_i has to be true.

Again, it's clear that the ILP instance can be constructed in polynomial time from ϕ. $\square$