CSE200: Complexity theory
NP and NP-completeness

Shachar Lovett

October 11, 2021

1 Decision vs search problems

Search problems are described by functions $f : \{0,1\}^* \to \{0,1\}^*$ which return an answer to a question, e.g. "what is the length of the shortest path between vertices s, t in a graph G?". **Decision problems** are described by functions $f : \{0,1\}^* \to \{0,1\}$ which return a YES/NO answer to a question, e.g. "is the length of the shortest path between vertices s, t at most ℓ". In nearly all cases, we can reduce search problems to decision problems (e.g. by binary search in this example).

So we will mainly restrict our attention from now on to decision problems. A decision problem can be equally described as **language**,

$$L = \{x \in \{0,1\}^* : f(x) = 1\}.$$

Let us recall some definitions. A language $L \subset \{0,1\}^*$ is **decideable** or **computable** if the function $f(x) = 1_{x \in L}$ is computable. For a time bound $T : \mathbb{N} \to \mathbb{N}$,

$$\mathrm{TIME}(T) = \{L \subset \{0,1\}^* : L \text{ is decideable in time } O(T)\}.$$

We have two standard definitions: P, polynomial time-solvable problems, which is what we think of as "efficiently solvable problems",

$$\mathrm{P} = \cup_{c \geq 1} \mathrm{TIME}(n^c).$$

EXP is the class of exponential-time solvable problems,

$$\mathrm{EXP} = \cup_{c > 0} \mathrm{TIME}(2^{n^c}).$$

2 The class NP

The class NP captures problems, where solutions can be verified in polynomial time.

Definition 2.1 (NP). *A language $L \subset \{0,1\}^*$ is in NP if there exists a polynomial time TM V (called a verifier) and a constant $c \geq 0$ such that*

$$L = \{x : \exists y, |y| \leq |x|^c, V(x,y) = 1\}$$

That is, $x \in L$ is there exist a "proof" or "witness" that is easy to verify (namely, can be verified by a poly-time TM). We think of V as a verifier for the fact that $x \in L$ given the proof. Note that since V runs in time at most $|x|^c$ we can assume that $|y| \leq |x|^c$.
Examples:

1. CLIQUE is the language of (G,k), where G is a graph with a clique of size at least k. CLIQUE is in NP since if G has n vertices, then a proof that G has a clique of size k is a list of the vertices in the clique. Given this list, it is simple to verify that they indeed form a clique.

2. Traveling Salesman Problem (TSP) is the language of (G,k) where G is a graph with weights, such that there exists a path covering all vertices in G with weights summing to at most k. TSP is in NP since such given such a path it is easy to verify that it satisfies the requirements.

3. Linear programming is the language of linear constraints which are satisfiable. It is in NP since a proof is a solution.

4. 0-1 linear programming is the language of linear constraints which are satisfiable by a 0-1 solution. It is in NP since a proof is a solution.

5. COMPOSITE is the language of numbers n which are composite (e.g non-prime). It is in NP since a proof is a decomposition $n = ab$, which can be verified in polynomial time.

Some of these problems are in P (linear programming, composite) while the others seem much harder.

Claim 2.2. *$P \subset NP \subset EXP$.*

Proof. P $\subset$ NP is obvious, since a verifier can ignore the proof. NP $\subset$ EXP since we can try all possible witnesses. Let $L \in$ NP. For any input $x \in L$ of length $|x| = n$, the possible witnesses are $y \in \{0,1\}^{n^c}$ for some $c > 0$. Hence, we can try all of them in time 2^{n^c} which is in EXP. □

3 NP completeness

It turns out that some languages are harder than others. This is formally defined by a reduction.

Definition 3.1 (Poly-time reduction)**.** *Language L_1 is poly-time reducible to a language L_2, if there exists a poly-time computable function $f : \{0,1\}^* \to \{0,1\}^*$, such that for all $x \in \{0,1\}^*$:*

$$x \in L_1 \iff f(x) \in L_2.$$

We denote this as $L_1 <_p L_2$.

Comment: this notion of a reduction is called a *mapping reduction* (also called Karp reduction). There is also a more general notion called a *Turing reduction* (also called Cook reduction), where in order to solve if $x \in L_1$ one can ask whether some $y \in L_2$ for several ys in an adaptive fashion. For now, we will only consider the first simpler notion.

Definition 3.2 (NP-complete languages)**.** *A language L is called **NP-hard** if for any $L' \in NP$ we have $L' <_p L$. If furthermore $L \in NP$ then we say L is **NP-complete**.*

Claim 3.3. *The following holds:*

1. *If $L_1 <_p L_2$ and $L_2 <_p L_3$ then $L_1 <_p L_3$.*
2. *If $L_1 <_p L_2$ and $L_2 \in P$ then $L_1 \in P$.*
3. *If $L_1 <_p L_2$ and L_1 is NP-hard then L_2 is NP-hard.*

Proof. The claim follows from the definitions:

1. We have $x \in L_1 \iff f(x) \in L_2$ and $x \in L_2 \iff g(x) \in L_3$ where f, g are poly-time computable. Hence $x \in L_1 \iff g(f(x)) \in L_3$ and the composed map $g(f(x))$ is poly-time computable.
2. We have $x \in L_1 \iff f(x) \in L_2$ where f is poly-time computable. Since L_2 is in P there is a poly-time TM M such that $x \in L_2 \iff M(x) = 1$. So, $x \in L \iff M(f(x)) = 1$, and $M(f(x))$ is poly-time computable.
3. If L_1 is NP-hard then for any language $L' \in$ NP, $L' <_p L_1$. By claim (1) we get $L' <_p L_2$ as well. So L_2 is NP-hard.

□

At this point, we know that NP-complete languages is a powerful concept, however it is unclear whether there are such languages. The following theorem shows that there are NP-complete languages.

Theorem 3.4 (Existence of NP-complete languages)**.** *The following language is NP-complete.*

$$L = \{(\langle M \rangle, x, 1^t) : \exists y \in \{0,1\}^*, |y| \leq t, M(x,y) = 1, M(x,y) \text{ accepts after at most } t \text{ steps}\}.$$

Proof. Lets first verify that $L \in \text{NP}$. For this, we need to build a poly-time verifier V for L. Define V as follows:

$$V(\langle M \rangle, x, 1^t, y) = \begin{cases} \text{accept} & \text{if } |y| \leq t, M(x,y) = 1, M(x,y) \text{ accepts after at most } t \text{ steps} \\ \text{reject} & \text{otherwise} \end{cases}.$$

That is, V verifies that y satisfies the requirements in L. Note that V runs by simulating M for at most t steps, and the input has length at least t, hence $V \in \text{P}$. This implies that $L \in \text{NP}$.

Next, we show that L is NP-Hard. Fix a language $L' \in \text{NP}$, and we will show that $L' <_p L$. By definition, there exists $c \geq 1$ and a TM $M \in \text{TIME}(n^c)$ such that

$$x \in L' \iff \exists y, |y| \leq |x|^c, M(x,y) \text{ accepts.}$$

Define the following function $f : \{0,1\}^* \to \{0,1\}^*$:

$$f(x) = (\langle M \rangle, x, 1^{|x|^c}).$$

Then $x \in L' \iff f(x) \in L$. Moreover, f can be computed in poly-time. □

4 Combinatorial NP-complete problems

Cook [1] and Levin [3] proved in the early 70s that there are "natural" combinatorial problems which are NP-complete. Shortly after, Karp [2] gave a list of 20+ well known combinatorial problems, all NP-complete. We describe some of them below.

Definition 4.1 (SAT)**.** *A CNF formula is a conjunction of disjunctions, e.g*

$$\varphi(x_1, x_2, x_3, x_4) = (x_1 \vee x_3 \vee \overline{x_4}) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \overline{x_4}).$$

Any function $\varphi : \{0,1\}^n \to \{0,1\}$ can be written as a CNF of size at most $n2^n$. A CNF formula is satisfiable if there exists an assignment to the variables making it true. We define

$$SAT = \{\textit{Satisfiable CNF formulas}\}.$$

Theorem 4.2 (Cook-Levin)**.** *SAT is NP-complete.*

Proof. Clearly SAT $\in$ NP since a proof that $\varphi \in$ SAT is an assignment to the variables of φ making it true. The harder claim is that SAT is NP-hard. Let $L \in \text{NP}$. Then there exists a TM $V \in \text{TIME}(n^c)$ verifying L. That is,

$$x \in L \iff \exists y, V(x,y) = 1.$$

We assume without loss of generality that V has a single tape with alphabet $\Gamma = \{0, 1, \perp\}$. The tape is initially initialized by the input, and then is used to perform the work.

Let $T = O(n^c)$ be a bound on the running time and space used by V. This means that only the first T cells in the tape are ever used. We define boolean variables capturing the contents of the tape at every step of the computation. For $1 \leq i, t \leq T$ and $\gamma \in \Gamma$ define:

- $TAPE^t_{i,\gamma} = 1$ if at the t-th time step, the i-th symbol in the tape equals γ.
- $HEAD^t_i = 1$ if at the t-th time step, the head is in the i-th cell.
- $STATE^t_q = 1$ if at the t-th time step, the state is q.

The theorem is based on the fact that every step of the computation is local.

Initialization. Let $x \in \{0,1\}^n$ be the input and $y \in \{0,1\}^t$ a potential witness. Define:

$$TAPE^0_{i,\gamma} = \begin{cases} 1 & \text{if } i = 1,\ldots,n \text{ and } \gamma = x_i \\ 1 & \text{if } i = n+1,\ldots,n+t \text{ and } \gamma = y_{i-n} \\ 0 & \text{otherwise} \end{cases}$$

$$HEAD^0_i = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$STATE^0_q = \begin{cases} 1 & \text{if } q \text{ is the start state of } V \\ 0 & \text{otherwise} \end{cases}$$

Consistency. The tape contents, head locations and state in step $t+1$ depend just on those on time t. Moreover, these are local computations. For example, the HEAD at time $t+1$ can only be at coordinate i only if the HEAD at time t was at one of the coordinates $\{i-1, i, i+1\}$.

Recall that the transition function of a TM is $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1,0,1\}$, where $\delta(q,\gamma) = (q',\gamma',d)$ if at state q and input character a, the TM writes γ' at the current location, changes the state to q' and moves in direction d in the tape (we denote the head movement here as $\{-1,0,1\}$ instead of $\{L,S,R\}$ for convenience).

So, for every $q \in Q, \gamma \in \Gamma, d \in \{-1,0,1\}$ and every $i = 1,\ldots,T$ we have the constraint:

$$\text{If } HEAD^t_i = 1, TAPE^t_{i,\gamma} = 1, STATE^t_q = 1 \text{ and } \delta(q,\gamma) = (q',\gamma',d) \text{ then}$$
$$HEAD^{t+1}_{i+d} = 1, TAPE^{t+1}_{i,\gamma'} = 1, STATE^{t+1}_{q'} = 1.$$

This is a formula on a **constant number of variables**, independent of the input size. So we can create a CNF formula expressing it. In addition, we need to add formulas that for every t, exactly one of $\{HEAD^t_i : i = 1,\ldots,T\}$ is 1; exactly one of $\{STATE^t_q : q \in Q\}$ is 1; and exactly one of $\{TAPE^t_{i,\gamma} : \gamma \in \Gamma\}$ is 1 for every i. All of these can be expressed by CNF formulas in the same way. Then, we just need to AND all the clauses and get the final CNF.

Output The output of the computation is $TAPE^T_{0,1}$.

Writing as a CNF. All the identities above can be written as a CNF of size poly(T). Then, requiring them all together to hold is done by AND-ing them, which will still be a polynomial size CNF. So, we can write a poly-size CNF $\varphi(x, y)$, such that $V(x, y) = 1 \iff \varphi(x, y) = 1$. Moreover, φ can be computed in time $O(T^2) = O(n^{2c})$. Let $\varphi_x(y) = \varphi(x, y)$ be the CNF formula obtained by plugging in the value of x. Then, we conclude that

$$x \in L \iff \exists y, V(x, y) = 1 \iff \exists y, \varphi(x, y) = 1 \iff \exists y, \varphi_x(y) = 1.$$

So $L <_p$ SAT by the reduction $f(x) = \varphi_x$. □

5 Proving NP-completeness by reductions

Once we showed that SAT is NP-complete, we can use this to show that many other combinatorial problems are also NP-complete. The general recipe is as follows: assume we know that a language L is NP-complete (say SAT) and we want to use this to prove that some other language L' is also NP-complete. We need to show two things:

1. L' is in NP. This is usually shown directly.
2. $L <_p L'$. That is, there is a poly-time computable function f which maps inputs of L to inputs of L', such that $x \in L \iff f(x) \in L'$.

Definition 5.1 (k-SAT)**.** *A k-CNF is a CNF where each clause has at most k variables.*

$$\textit{k-SAT} = \{\textit{Satisfiable k-CNF formulas}\}.$$

Theorem 5.2. *3SAT is NP-complete (and hence also 4SAT, 5SAT, etc).*

Proof. We need to show that every CNF formula can be reduced to 3-CNF formula, with only a polynomial blow up in the size, and where the reduction can be computed by a poly-time TM. Let $\varphi(x) = C_1(x) \wedge \ldots \wedge C_m(x)$ be a CNF formula, where $|x| = n$ and $m = poly(n)$. It suffices to change each clause to a 3-CNF. Let $C(x)$ be a clause, and assume w.l.o.g that

$$C(x) = v_1 \vee v_2 \vee \ldots \vee v_k,$$

where v_i is either a variable or its negation. Define new variables $y_1, \ldots, y_{k-1}$ given by

$$y_1 = v_1 \vee v_2, \quad y_2 = y_1 \vee v_3, \ldots, \quad y_{k-1} = y_{k-2} \vee v_k$$

Then $C(x) = y_{k-1}$, and we can express each of these formulas by a 3-CNF since they involve just 3 variables. To summarize, we transformed a CNF formula $\varphi(x)$ to a 3-CNF formula $\psi(x, y)$, where φ is satisfiable iff ψ is. Note that the process of creating ψ given a description of φ is done in poly-time, and hence SAT $<_p$ 3SAT. This shows 3SAT is NP-hard. Clearly 3SAT is also in NP, hence it is NP-complete. □

It turns out that 2SAT is not NP-complete. In fact, it is in P. So there is a dichotomy: k-SAT for $k = 2$ is in P, and for $k \geq 3$ is NP-complete.

Theorem 5.3. *2SAT is in P.*

Proof. Let $\varphi(x)$ be a 2-CNF on n variables. Any clause in φ has at most two literals (variables or their negations). Lets assume any clause in φ has exactly two literals, e.g. by replacing x_1 with $x_1 \vee x_1$. We create a directed graph G with vertices $V = \{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$. The edges of G are as follows: for a clause is of the form $v_i \vee v_j$, it can be equivalently written as

$$\neg v_i \to v_j \quad \text{or} \quad \neg v_j \to v_i$$

Which means "if $\neg v_i = 1$ then we must have $v_j = 1$", and similarly for the second term. We will record these by adding the edges $\neg v_i \to v_j$ and $\neg v_j \to v_i$ to G.

Next, we decompose G to strongly connected components. To recall the definition, a strongly connected component is a set of vertices $S \subset V$ such that for any two vertices $u, v \in S$ there is a directed path from u to v and from v to u. Lets order the set of strongly connected components $S_1, \ldots, S_m$ so that edges from $v \in S_i$ only go to $u \in S_j$ for $j \geq i$. This can be done via putting the strongly connected components in a DAG. All this pre-processing can be done in polynomial time.

Next, if for some variable both x_i and $\neg x_i$ are in the same strongly connected component then φ is unsatisfiable, since if x is a satisfying assignment and $x_i = 1$ then, by following the path from x_i to $\neg x_i$ we get that $\neg x_i = 1$, e.g. $x_i = 0$; and similarly if $x_i = 0$.

Otherwise, we will show that φ is satisfiable, by finding a satisfying assignment. Perform the following process: starting from S_m and going backwards, if x_i is an unassigned variable appearing for the first time then set $x_i = 1$; if $\neg x_i$ appears for the first time assign $x_i = 0$. The order inside the strongly connected components doesn't matter as we assume that $x_i, \neg x_i$ never appear in the same component.

Lets see why the assignment we get is satisfying. Assume $v_i \vee v_j$ is a clause in φ which is not satisfied by the assignment. This means we set $v_i = v_j = 0$. This means we found $\neg v_i$ before v_i and $\neg v_j$ before v_j. Note that $\neg v_i, \neg v_j$ can't both be in the same component since otherwise, the edge $\neg v_i \to v_j$ implies that there is a path $\neg v_j \rightsquigarrow \neg v_i \to v_j$, so v_j should have appeared before $\neg v_j$ in our process. Assume without loss of generality that we encounter $\neg v_i$ at a prior component to $\neg v_j$. However, as $\neg v_i \to v_j$ it means that we would encounter v_j before $\neg v_j$, e.g. setting $v_j = 1$. □

Let us now prove that another famous problem is NP-complete - finding if a graph has a large clique.

Theorem 5.4. *CLIQUE is NP-complete.*

Proof. CLIQUE is clearly in NP. We need to show CLIQUE is NP-hard. We will reduce 3SAT to CLIQUE. Let φ be a 3-CNF, where we assume w.l.o.g that each clause has three variables. Assume φ has n variables and m clauses. We define a graph G on $7m$ vertices. For each clause of G there will be 7 vertices. For example, for $C(x) = x_1 \vee \neg x_2 \vee x_3$ these will

correspond to the 7 assignments to x_1, x_2, x_3 which make C true. We connect two vertices by an edge if they don't share a variable and its negation (in other words, the two assignments they define are consistent).

We next claim φ is satisfiable iff G has a clique of size m. If φ is satisfiable, let x be satisfying assignment. For each clause choose the vertex corresponding to x. They form a clique of size m. On the other hand, any clique in G can contain at most one vertex for each clause, hence a clique of size m contains exactly one vertex for each clause; these specify a consistent assignment to the variables which satisfy all the clauses. □

Linear programming is the problem of finding solutions for a system of linear inequalities. Famously, linear programming is solvable in poly-time, which is the basis for many optimization algorithms. 0-1 programming is a linear programming with the additional constraint that the solutions are all 0 or 1. We next show it is NP-complete.

Theorem 5.5. *0-1 PROGRAMMING is NP-complete.*

Proof. 0-1 PROGRAMMING is clearly in NP, as we can verify a solution easily. To show it is NP-complete we reduce 3SAT to 0-1 PROGRAMMING. To do so, we represent each clause as a set of linear inequalities. This is best explained via an example: to represent say the clause $x_1 \vee \neg x_2 \vee x_3$ in the 3-CNF, we add the constraint $x_1 + (1 - x_2) + x_3 \geq 1$. □

We covered only a small fraction of the known NP-complete problems. Some more famous examples are:

- Checking if a graph can be colored with c colors, for any constant $c \geq 3$.
- Checking if a graph has an hamiltonian cycle, or an hamiltonian path (hamiltonian cycle, hamiltonian path).
- Checking if a graph has a vertex cover of a given size (vertex cover).
- Computing the maximal cut for a graph (max cut).
- The Traveling Salesman Problem (TSP).
- Checking if given a list of numbers and a target, there is a subset of these numbers that sums to the desired target (subset sum).
- Finding optimal way to pack a list of numbers into buckets, each with a bounded capacity (bin packing).

References

[1] S. A. Cook. The complexity of theorem-proving procedures. In Proceedings of the third annual ACM symposium on Theory of computing, pages 151–158. ACM, 1971.

[2] R. M. Karp. Reducibility among combinatorial problems. In Complexity of computer computations, pages 85–103. Springer, 1972.

[3] L. A. Levin. Universal sequential search problems. Problemy peredachi informatsii, 9(3):115–116, 1973.