P, NP, Poly-time Reductions, and Satisfiability

Reading: Sipser chapter 7, in particular 7.3.

1 Polynomial Time Reducibility

In earlier lectures we studied reductions. We had that if a problem A reduces to another problem B, then a solution for B can be used to solve A. We will now refine this definition to take computational efficiency into account.

Definition 1 (Polynomial time computable function)**.** *A function $f : \{0,1\}^* \to \{0,1\}^*$ is a polynomial time computable function if there exists a polynomial time TM M that halts with just $f(w)$ on the tape, when started on input w.*

Definition 2 (Polynomial time mapping reducible; aka polynomial time many-one reducible)**.** *A language A is polynomial time (mapping) reducible to language B, written $A \leq_p B$, if there exists a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$, such that for every $x \in \{0,1\}^*$ we have that*

$$x \in A \iff f(x) \in B$$

The function f is called the polynomial time reduction of A to B.

This is the simplest form of reducibility. Also, it is possible that multiple instances of A get mapped to the same instance of B.

Theorem 1.1. *If $A \leq_p B$ and $B \in P$, then $A \in P$.*

Proof. Let M be a polynomial time algorithm that decides B and f the polynomial time reduction from A to B. We can get a polynomial time algorithm N for solving A, that on input x works as follows:

- Compute $f(x)$.
- Run M on input $f(x)$ and give the same output as M.

Then we have that $x \in A$ if and only if $f(x) \in B$. Since both steps run in polynomial time in the length of x, we obtain that the two steps together also run in polynomial time in $|x|$. □

2 Satisfiability

We now introduce satisfiability problems, which are related to how a computer circuit actually works (with AND and OR operations, for example).

A *literal* is a Boolean variable or a negated Boolean variable, such as x or $\overline{x}$. A *clause* is several literals connected with ORs, such as $(x_1 \vee \bar{x_2} \vee x_3 \vee \bar{x_4})$. A *Boolean formula* is in *conjunctive normal form* (cnf-formula) if it consists of multiple clauses connected with ANDs, such as the next example.

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6})$$

Definition 3 (Satisfiability Problem)**.** *The Satisfiability Problem (SAT) consists of determining if a given formula is satisfiable:*

$$SAT = \{\langle \phi \rangle \mid \phi \textit{ is a satisfiable CNF-formula}\}.$$

Note that for a formula with n variables, the number of possible assignments is exponential in n (two choices for every variable, n variables, so 2^n choices).

We will look also at two restricted versions of SAT, namely $3SAT$ and $2SAT$, which correspond to satisfiability for 3cnf and 2cnf formulas, respectively. A *3cnf-formula* consists of clauses with three literals:

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

Similarly, a *2nf-formula* is defined as a cnf-formula with clauses consisting of two literals.

Our first polynomial time reduction will involve showing that $3SAT$ is reducible to the $CLIQUE$ problem, which is defined by

$$CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a k-clique}\}$$

Theorem 2.1. *$3SAT$ is polynomial time reducible to $CLIQUE$.*

Proof. Let ϕ be a $3CNF$ formula with k clauses:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \quad \cdots \quad \wedge (a_k \vee b_k \vee c_k).$$

Given ϕ, we will construct a polynomial time reduction f that will generate a string $\langle G, k \rangle$, where G is a graph for which we must find a k-clique or report that none exists. The graph G_k has k groups of nodes, one for each clause in the formula ϕ, that we called triplets $t_1 \dots t_k$. Each node in a triplet t_i corresponds to a literal in the clause that the triplet t_i is associated with. We label each node with the corresponding literal as illustrated in Figure 1.

We show that ϕ is satisfiable if and only if the graph obtained has a k clique. Suppose that ϕ has a satisfying assignment. Pick one true literal ℓ_i in each clause i. Then we can obtain a k-clique by selecting for each literal ℓ_i the node in G corresponding to that literal. We are assured

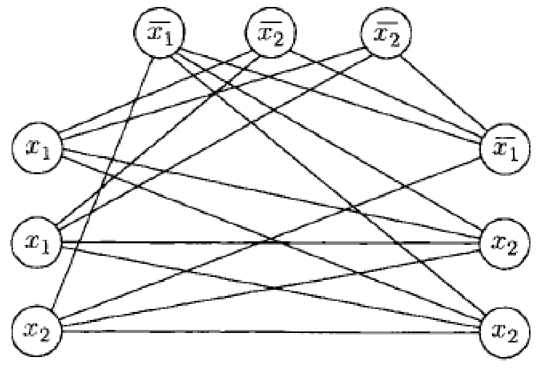

Figure 1: Graph obtained for the formula $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2} \vee \bar{x_2}) \wedge (\bar{x_1} \vee x_2 \vee x_2)$.

to have an edge between every two nodes selected because we did not pick any nodes inside the same triplet, and we didn't pick two nodes corresponding to a literal and its negation either.

For the other direction, if the graph G has a k-clique C, then we assign true to the literals corresponding to the nodes in C. Since there is an edge between every two nodes u, v in C, it cannot be the case that u and v represent a literal and its negation, or are inside the same triplet. Thus we can label each node in C by "true", thus assigning true to the literals they represent. This is a satisfying assignment for ϕ as required.

Thus we found a polynomial time computable function f that maps 3cnf-formulas to instances of clique, such that the formula is satisfiable if and only if the clique instance has a solution. □

We can convert any cnf formula to a 3cnf formula in polynomial time as follows.

Proposition 1. *There exists a polynomial time algorithm that takes a cnf formula and converts it to a 3cnf formula.*

Proof. (sketch) We illustrate the idea of the construction using as example the clause $(x_1 \vee x_2 \vee x_3 \vee x_4)$. This clause has length > 3, so we add a dummy variable to capture the $3rd$ and $4th$ literals of the clause; let $y = x_3 \vee x_4$. Replace the clause by $(x_1 \vee x_2 \vee y) \wedge (\bar{y} \vee x_3 \vee x_4)$.

Thus for each clause that has length greater than 3, we iteratively introduce dummy variables until the length of all the clauses have length at most 3. □

Theorem 2.2. *CLIQUE is polynomial time reducible to* $3SAT$.

Proof. Given a graph $G = (V, E)$ and a number k, create variables $x_{i,v}$ for every $1 \leq i \leq k$ and every $v \in V$, with the interpretation that $x_{i,v} = True$ if node v is the i-th vertex in the clique. We encode the conditions:

1. For each i, there is an ith vertex in the clique: $\vee_{v \in V} x_{i,v}$.

2. For all i, j, the ith vertex is different from the jth vertex: for each $v \in V$, $\bar{x}_{i,v} \vee \bar{x}_{j,v}$.

3. For each non-edge $(u, v) \notin E$, u and v cannot both belong to the clique: for each i, j, $\bar{x}_{i,u} \vee \bar{x}_{j,v}$.

Taking all the constraints together, we obtain a cnf formula that is satisfiable if and only if the graph has a k-clique. In order to get a $3cnf$, we apply the polynomial time reduction given by Proposition 1. $\square$

Before exploring $3SAT$ in more depth, let's pause for a moment and consider the variants of kSAT for $k < 3$. For $k = 1$, a formula is simply a conjunction, such as $\phi = x_1 \wedge \bar{x_1} \wedge \ldots \wedge x_k$. This is easily solvable in polynomial time by checking that we don't have both a literal and its negation in the formula. What about $k = 2$? As we will see next, $2SAT$ is solvable in polynomial time too.

Theorem 2.3. *$2SAT$ is solvable in polynomial time.*

Proof. We will prove this by reducing $2SAT$ to a graph theoretic problem in P. Given a $2SAT$ formula ϕ, create a directed graph G with $2n$ vertices, such that each vertex corresponds to a true or not true literal for each variable in ϕ. For each clause $a \vee b$ in ϕ, add an edge from $\bar{a}$ to b and one from $\bar{b}$ to a.

For example, the $2CNF$ $\phi = (\bar{x} \vee y) \wedge (\bar{y} \vee z) \wedge (x \vee \bar{z}) \wedge (z \vee y)$ has the graph in Figure 2.

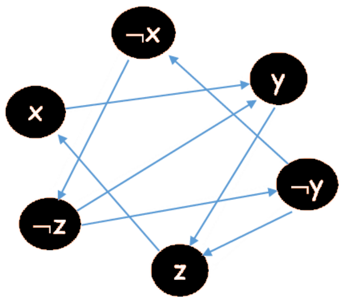

Figure 2: Graph constructed for the 2cnf formula $(\bar{x} \vee y) \wedge (\bar{y} \vee z) \wedge (x \vee \bar{z}) \wedge (z \vee y)$.

Lemma 1. *If the graph G contains a directed path from a to b, then it also contains a path from $\bar{b}$ to $\bar{a}$.*

Proof. Suppose there is a directed path $P = a \to c_1 \to \ldots \to c_k \to b$. Then by construction of the graph, if there is an edge from x to y, there is also an edge from $\bar{y}$ to $\bar{x}$. Then for every edge $i \to j$ in the path, the graph also contains the edge $\bar{j} \to \bar{i}$, which gives the desired path $P' = \bar{b} \to \bar{c_k} \to \ldots \bar{c_1} \to \bar{a}$. $\square$

Lemma 2. *The 2cnf formula ϕ is* unsatisfiable *if and only if there exists a variable x such that the following two conditions are met in the graph G:*

- *there is a path from x to $\bar{x}$.*
- *there is a path from $\bar{x}$ to x.*

Proof. This can be proved by contradiction and left as an exercise. $\square$

Thus we can decide if a formula is satisfiable by verifying that for each literal x the graph G does not contain paths from x to $\bar{x}$ and viceversa. This can be done in polynomial time, so we can decide $2cnf$ in polynomial time.

In order to actually find a satisfying assignment, the arguments above imply the following algorithm:

1. Given 2cnf formula ϕ, construct graph G as outlined and check if ϕ is satisfiable.
2. If ϕ is not satisfiable, return False.
3. Else, pick an unassigned literal a, with no path from a to $\bar{a}$, and assign it $True$.
4. Assign $True$ to each vertex (i.e. literal) reachable from a and assign $False$ to their negations.
5. Go back to step 3 if any vertices are still unassigned.

□

Can't we convert a cnf formula to a 2cnf formula, and then solve the latter in polynomial time? It turns out that the answer is "probably not". Consider the next example.

Example 1. *Let $\phi = (a \vee b \vee c)$ be a 3cnf formula with one clause. Consider the method of introducing dummy variables we used in Proposition 1. Let the dummy variable be $y = b \vee c$. The result is $(a \vee y) \wedge (\bar{y} \vee b \vee c)$. We still ended up with a clause of length 3. Does it help to introduce a second dummy variable $z = b \vee c$? Then we get $(a \vee y) \wedge (\bar{y} \vee z) \wedge (\bar{z} \vee b \vee c)$, so we still have a clause of length 3.*

There are other methods for getting around this issue of introducing dummy variables this way, but the resulting 2cnf formulas are exponential in the size of the original cnf formula.

3 NP-completeness

There exist problems in NP such that if any of these problems are solvable in polynomial time, then every problem in NP is solvable in polynomial time. These are the hardest problems in NP and are called *NP-complete.*

Definition 4 (NP-complete)**.** *A language L is NP-complete if*

- *$L \in NP$ and*
- *Every language $A \in NP$ is polynomial time reducible to L.*

Theorem 3.1. *If a language A is NP-complete and $A \in P$, then $P = NP$.*

Proof. The proof is straightforward by the definition of polynomial time reducibility. □

Theorem 3.2. *If a language A is NP-complete and $A \leq_p B$ for $B \in NP$, then B is NP-complete.*

Proof. Since A is NP-complete, every language $L \in NP$ reduces to A in polynomial time. Given such a language L, let f be the poly-time computable function that reduces L to A. Suppose g is the polynomial time computable function that reduces A to B. Then the composition gives a poly-time reduction from L to B. □

In some instances, we will be dealing with problems that only satisfy the second condition in the definition of NP-completeness. These problems will be called NP-hard.

Definition 5 (NP-hard)**.** *A language L is NP-hard if every language $A \in NP$ is polynomial time reducible to it.*

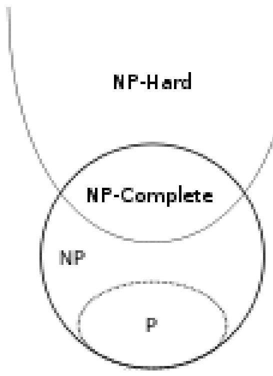

Figure 3: Relations between complexity classes.