1 P vs NP

Is P = NP?

Can the collection of problems that have short, efficiently checkable proofs, be the same as the collection of problems for which you can *find* short, efficiently checkable proofs, *efficiently*?

P versus NP

- Are there problems in NP that are not in P?
- If there are, then the *most difficult* problems in NP must be such problems.
- How do we define "most difficult"?
  - Reductions!

1.1 Reductions

Polynomial Time Reductions
Capturing the Relative Difficulty of Problems

Definition 1. A *polynomial time reduction* from L_1 to L_2 is a *polynomial time computable* function $f : \Sigma^* \to \Sigma^*$ such that

$$u \in L_1 \text{ iff } f(u) \in L_2$$

L_1 is said to be *polynomial time reducible* to L_2 and is denoted by $L_1 \ \leq_P \ L_2$.

Properties of Reductions

Proposition 2. *If $L_1 \ \leq_P \ L_2$ and $L_2 \ \leq_P \ L_3$ then $L_1 \ \leq_P \ L_3$*

Proof. If f is a polynomial time reduction from L_1 to L_2 running in time n^k and g is a polynomial time reduction from L_2 to L_3 computed in time n^ℓ then $g \circ f$ is a reduction from from L_1 to L_3 and can be computed in time $O(n^k + (n^k)^\ell) = O(n^{k\ell})$. □

Proposition 3. *If $L_1 \ \leq_P \ L_2$ and $L_2 \in P$ then $L_1 \in P$.*

Proof. Let f be the reduction from L_1 to L_2 (running in time n^k) and let B be a polynomial time algorithm deciding L_2 (in time n^ℓ). Then the algorithm for L_1 on input w, computes $f(w)$ and runs B on $f(w)$. The total running time is $O(n^k + (n^k)^\ell) = O(n^{k\ell})$. □

1.2 Completeness

Completeness
Hardest Problems in a Class

Definition 4.
- L is said to be *NP-hard* iff for every $L' \in$ NP, $L' \leq_P L$
- L is said to be *NP-complete* iff $L \in$ NP and L is NP-hard

---

2 Examples

2.1 SAT

Propositional Logic

Formulas in propositional logic are

- built from propositions,
- using $\wedge$ (conjunction), $\vee$ (disjunction), and $\neg$ (negation).

Example 5. Examples of formulas are $(p \vee (\neg p))$, $((p \wedge q) \vee (\neg p) \vee (\neg q))$, and $((\neg p) \vee q)$.

Conjunctive Normal Form Formulas

Definition 6.
- A *literal* is a propositional variable p or its negation $\neg p$.
- A *clause* is a disjunction of literals. Example, $p \vee (\neg q) \vee r$.
- A formula is said to be in *conjunctive normal form* (CNF) if it is a conjunction of clauses. Example, $((p \vee (\neg q)) \wedge ((\neg p) \vee q))$

Proposition 7. *Every formula in propositional logic is equivalent to a formula in conjunctive normal form.*

Proof. Push all the negations inside using De Morgan laws, and then distribute the disjunctions over the conjunctions. ☐

Satisfiable Formulas

Definition 8. A formula φ is *satisfiable* if there is a assignment to the propositions such that φ evaluates to true. φ is *unsatisfiable* if it is not satisfiable.

Example 9. $(p \vee (\neg q)) \wedge ((\neg p) \vee q)$ is satisfiable because it evaluates to 1 (true) when $p \mapsto 1$ and $q \mapsto 1$.

$(p \wedge (\neg p))$ is unsatisfiable.

Satisfiability Problem

SAT

SAT $= \{\langle\varphi\rangle \mid \varphi$ is a conjunctive normal form formula that is satisfiable$\}$

Definition 10. A *k-CNF* formula is a formula φ in conjunctive normal form such that every clause in φ has exactly k literals.

kSAT

kSAT $= \{\langle\varphi\rangle \mid \varphi$ is a k-CNF formula that is satisfiable$\}$

SAT $\in$ NP

Proposition 11. *SAT $\in$ NP*

Proof. SAT is polynomially verifiable. The proof that $\langle\varphi\rangle \in$ SAT is a satisfying assignment σ. Observe that $|\sigma|$ is equal to the number of propositions in φ, and given an assignment σ, one can check in $O(|\varphi|)$ time if φ by evaluating each of subformulas starting from the literals.

Another proof would be to give a nondeterministic algorithm. The algorithm guesses a truth assignment σ, and checks if φ evaluates to true under σ. The running time is polynomial because of reasons listed in the previous paragraph. $\square$

Cook-Levin Theorem

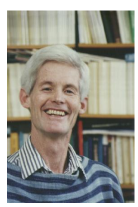

Figure 1: Stephen A. Cook

Figure 2: Leonid Levin

Theorem 12 (Cook-Levin)**.** *3SAT is NP-hard.*

Proof. Not enough time to cover. □

Corollary 13. *3SAT is NP-complete.*

Corollary 14. *SAT is NP-complete.*

Proof. We have already established that SAT $\in$ NP. We also know (from Cook-Levin Theorem) that for every $L \in$ NP, we have $L \leq_P$ 3SAT. It is easy to see that 3SAT $\leq_P$ SAT: since 3SAT is a special case of SAT, the reduction on input φ returns φ, if φ is a 3-CNF formula. Finally, since reductions compose, we have for every $L \in$ NP, $L \leq_P$ SAT, and so SAT is NP-hard. Hence, we have SAT is NP-complete. □

Recipe for Proving NP-hardness

To prove that A is NP-hard, we need to show that for every $L \in$ NP, $L \leq_P A$.

- Suppose B is NP-hard and $B \leq_P A$.
- Then, since for every $L \in$ NP, $L \leq_P B$ (NP-hardness of B), and reductions compose, we have established the NP-hardness of A.

2.2 Independent Set

Independent Set

Definition 15. Given graph $G = (V, E)$, $I \subseteq V$ is an *independent set* iff for every $u, v \in I$, $(u, v) \notin E$, i.e., it is subset of vertices no two of which are joined by an edge.

Example 16.

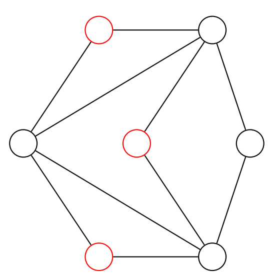

Figure 3: An independent set is shown in red

Independent Set Problem

Definition 17. INDEP $= \{\langle G, k\rangle \mid G$ is a graph that has an independent set of size at least $k\}$

Theorem 18. *INDEP is NP-complete.*

Proof. First observe that INDEP $\in$ NP. The nondeterministic algorithm does the following. If k is more than the number of vertices in G, it answers "no". Otherwise, it guesses an independent set of size k, and checks that no two vertices in the (guessed) set have an edge between them. This runs in time that is $O(|G|)$.

To prove hardness, we will show that 3SAT $\leq_P$ INDEP. That is given a 3-CNF formula φ, the reduction will (in polynomial time) construct a graph G_φ and number k_φ such that $\varphi \in$ 3SAT iff $\langle G_\varphi, k_\varphi \rangle \in$ INDEP. There are two ways to think about 3SAT

- Find a way to assign 0/1 to the variables such that the formula evaluates to true
- Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in *conflict*, i.e., you pick x_i and $\neg x_i$

We will take the second view of 3SAT to construct the reduction.

The informal overview of the reduction is as follows

- G_φ will have one vertex for each literal in a clause.
- Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
- Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
- Take k_φ to be the number of clauses

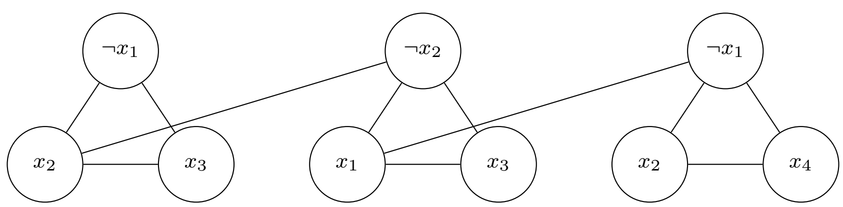

Figure 4: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Observe that the reduction can be computed in polynomial time. To establish the correctness of the reduction we need to show that G_φ has an independent set of size k_φ iff φ is satisfiable. Suppose I is an independent set of size k_φ (= the number of clauses in φ).

- I must contain exactly one vertex from each clause.
- I cannot contain vertices labelled by conflicting clauses.
- Thus, it is possible to obtain a truth assignment that makes in the literals in S true; such an assignment satisfies one literal in every clause.

On the other hand suppose a is a satisfying truth assignment for φ. Then construct I as follows: pick one vertex, corresponding to true literals under a, from each triangle. I is an independent set of the appropriate size in G_φ. $\square$

2.3 Vertex Cover

Vertex Cover

Definition 19. Given a graph $G = (V, E)$, a *vertex cover* $C \subseteq V$ is a subset of vertices such that for every edge $e \in E$ at least one of its endpoints is in C.

Example 20.

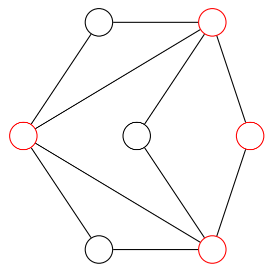

Figure 5: A vertex cover is shown in red

Vertex Cover and Independent Set

Proposition 21. *Let $G = (V, E)$ be a graph. I is an independent set iff $V \setminus I$ is a vertex cover.*

Proof. ($\Rightarrow$) Let I be any independent set

- Consider some edge $(u, v) \in E$
- Since I is an independent set, either $u \notin I$ or $v \notin I$
- Thus, either $u \in V \setminus I$ or $v \in V \setminus I$
- $V \setminus I$ is a vertex cover

($\Leftarrow$) Let $V \setminus I$ be some vertex cover

- Consider $u, v \in I$
- (u, v) is not edge, as otherwise $V \setminus I$ does not cover (u, v)
- I is thus an independent set $\square$

Vertex Cover Problem

Definition 22. $\text{VC} = \{\langle G, k\rangle \mid G \text{ is a graph that has a vertex cover of size at most } k\}$

Theorem 23. *VC is NP-complete.*

Proof. First observe that VC $\in$ NP. The nondeterministic algorithm guesses a vertex cover of size at most k, and checks that every edge has at least one of its endpoints in the (guessed) set. This runs in time that is $O(|G|)$.

To prove hardness, we will show that INDEP $\leq_P$ VC. Given a graph G with n vertices, the reduction f on input $\langle G, k\rangle$ will return $\langle G, n-k\rangle$. This is correct because our earlier observations show that G has a independent set of size at least k iff G has a vertex cover of size at most $n-k$. The reduction clearly can be computed in polynomial time. $\square$

Big Picture for the last time