PART 4:

Complexity Theory:

Complexity
Time Complexity Classes
Space Complexity Classes

Time Complexity Classes:

P, NP,
NP-completeness,
Polynomial-Time Reduction,
EXPTIME

The Complexity Class P

P =

{ Problems Solvable in Polynomial Time by DTMs }

Measuring Time Requirements

***timereq*(*M*):**

If *M* is a ***deterministic*** Turing Machine that halts on all inputs, then:

> ***timereq*(*M*)** = *f*(*n*) = the maximum number of steps that *M* executes on any input of length *n*.

The Language Class P

All and only languages that are decidable by a DTM in polynomial time!

$L \in$ P iff

- there exists some deterministic Turing machine M that decides L, and
- $timereq(M) \in \mathcal{O}(n^k)$ for some k.

"**Deterministic Polynomial-Time Deciding**"

We'll say that L is ***tractable*** iff it is in P!

Most Tractable Problems

Most tractable problems, i.e. problems in P can be solved

- *no more than* $\mathcal{O}(n^3)$ on conventional computers.
- *no more than* $\mathcal{O}(n^{16})$ on a one-tape DTM.

Closure of P under Complement

If a language L is in P, so is its complement $\neg$L!

Theorem 28.1: The class P is closed under complement.

Proof:

If *M* accepts *L* in polynomial time, swap accepting and non accepting states to accept $\neg L$ in polynomial time.

Languages That Are in P

- Every regular language in $\mathcal{O}(n)$ time.
- Every context-free language since there exist context-free parsing algorithms that run in $\mathcal{O}(n^3)$ time.
- Some languages that are not context free. $A^nB^nC^n$ in $\mathcal{O}(n^2)$ time.

Equivalence of Multi-tape and One-tape TMs

Theorem 17.1: Let M be a k-tape Turing machine for some $k \geq 1$. Then there is a standard TM M' where $\Sigma \subseteq \Sigma'$, and:

- On input x, M halts with output z on the first tape iff M' halts in the same state with z on its tape.
- On input x, if M halts in n steps, M' halts in $\mathcal{O}(n^2)$ steps.

Proof: Proof by Construction

Simulating a Real Computer by a Multi-tape TM

Theorem 17.4 A random-access, stored program computer can be simulated by a 7-tape Turing Machine. If the computer requires *n* steps to perform some operation, **the 7-tape Turing Machine simulation** will require $\mathcal{O}(n^3)$ steps.

Proof Idea:

Proof by Construction.

simcomputer will use 7 tapes:

Simulating a Real Computer by a One-tape TM

Theorem 17.4 A random-access, stored program computer can be simulated by a Turing Machine. If the computer requires n steps to perform some operation, **the Turing Machine simulation** will require $\mathcal{O}(n^6)$ steps.

Proof Idea:

Proof by Construction.

To Show That a Language Is In P

- ✓ State an algorithm that runs on a conventional computer.
- ✓ Describe a multi-tape, deterministic Turing Machine.
- ✓ Describe a one-tape, deterministic Turing Machine.

Regular Language is in P

Theorem 28.2 Every *regular language* can be decided in *linear time*. So every regular language is in P.

Proof Idea:

If L is regular, there exists some DFSM M that decides it.

Construct a deterministic TM M' that simulates M, moving its read/write head one square to the right at each step. When M' reads a ❑, it halts. If it is in an accepting state, it accepts; otherwise it rejects.

On any input of length n, M' will execute $n + 2$ steps.
So *timereq*(M') $\in \mathcal{O}(n)$.

Context-Free Language is in P

Theorem 28.3 Every *context-free language* can be decided in $\mathcal{O}(n^{18})$ *time*. So every context-free language is in P.

Proof Idea:

The Cocke-Kasami-Younger (CKY) algorithm can parse any context-free language in time that is $\mathcal{O}(n^3)$ if we count operations on a conventional computer.

That algorithm can be simulated on a standard, one-tape Turing machine in $\mathcal{O}(n^{18})$ steps.

CONNECTED is in P

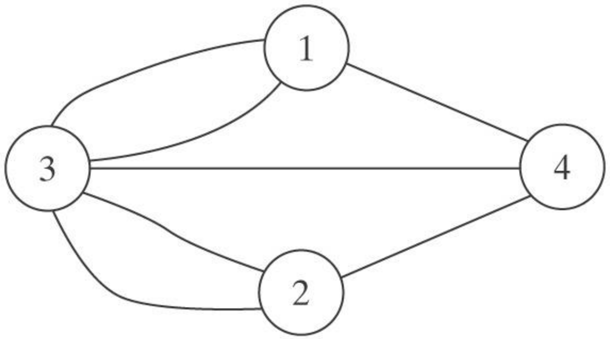
CONNECTED =
{<*G*> : *G* is an undirected graph and *G* is connected}.

Theorem 28.4

Eulerian Paths and Circuits

- An *Eulerian path* through a graph *G* is a path that traverses *each edge* in *G exactly once*.

- An *Eulerian circuit* through a graph *G* is a path that starts at some vertex *s*, ends back in *s*, and traverses each edge in *G* exactly once.



Eulerian Paths and Circuits

- A connected graph possesses an Eulerian path that is not a circuit iff it contains exactly two vertices of odd degree.

  Those two vertices will serve as the first and last vertices of the path.

- A connected graph possess an Eulerian circuit iff all its vertices have even degree.

EULERIAN-CIRCUIT is in P

EULERIAN-CIRCUIT = {<*G*> : *G* is an undirected graph, and *G* contains an Eulerian circuit} is in P.

Theorem 28.5

Spanning Trees

A *spanning tree* *T* of a graph *G* is a subset of the edges of *G* such that:

- *T* contains no cycles and
- Every vertex in *G* is connected to every other vertex using just the edges in *T*.

An unconnected graph has no spanning trees.

A connected graph *G* will have at least one spanning tree; it may have many.

Minimum Spanning Trees

A *weighted graph* is a graph that has a weight associated with each edge.

If *G* is a weighted graph, the *cost* of a tree is the sum of the costs (weights) of its edges.

A tree *T* is a *minimum spanning tree* of *G* iff:

- it is a spanning tree and
- there is no other spanning tree whose cost is lower than that of *T*.

MST is in P

MST = {<*G*, *cost*> : *G* is an undirected graph with a positive cost attached to each of its edges and there exists a minimum spanning tree of *G* with total cost less than *cost*} is in P.

Theorem 28.6

Primality Testing is in P

RELATIVELY-PRIME = {<*n*, *m*> : *n* and *m* are integers that are relatively prime} is in P.

PRIMES = {*w* : *w* is the binary encoding of a prime number} *is in P.*

COMPOSITES = {*w* : *w* is the binary encoding of a nonprime number} is in P.

Hamiltonian Path and Circuit

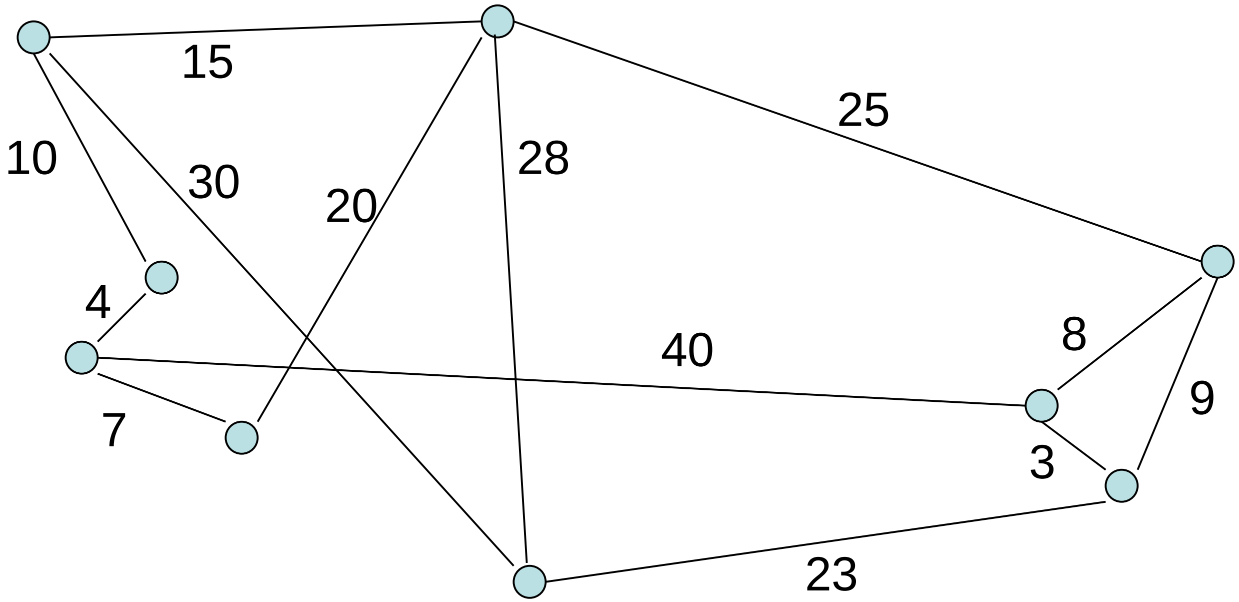
HAMILTONIAN-PATH = {<*G*> : *G* is an undirected graph and *G* contains a Hamiltonian path}.

HAMILTONIAN-CIRCUIT = {<*G*, *s*> : *G* is an undirected graph and *G* contains a Hamiltonian circuit}.

- A Hamiltonian path is a path that visits *each vertex exactly once*.
- A Hamiltonian circuit is a Hamiltonian path that starts some vertex and ends in the same vertex.

TSP is in P???

TSP-DECIDE = {<*G*, *cost*> : <*G*> encodes an undirected graph with a positive distance attached to each of its edges and *G* contains a Hamiltonian circuit whose total cost is less than <*cost*>} is in P???



TSP and Other Problems Like It

TSP-DECIDE, and other problems like it, share three properties:

1. The problem can be solved by searching through a space of partial solutions (such as routes). The size of this space grows exponentially with the size of the problem.

2. No better (i.e., not based on search) technique for finding an exact solution is known.

3. But, if a proposed solution were suddenly to appear, it could be checked for correctness very efficiently.

The Complexity Class NP

NP =

{ Problems Solvable in Polynomial Time by NDTMs }

Measuring Time Requirements

***timereq*(*M*):**

If *M* is a *nondeterministic* Turing Machine <u>all</u> of whose computational paths halt on <u>all</u> inputs, then:

***timereq*(*M*)** = $f(n)$ = the number of steps <u>on the longest path</u> that *M* executes on <u>any</u> input of length *n*.

The Language Class NP

All and only languages that are decidable by a NDTM in polynomial time!

$L \in$ NP iff:

- there is some **N**DTM M that **decides** L, and
- $timereq(M) \in \mathcal{O}(n^k)$ for some k.

"**Nondeterministic Polynomial-Time Deciding**"

Deterministic Verifying

A Turing Machine *V* is a *verifier* for a language *L* iff:

$$w \in L \text{ iff } \exists c\, (\langle w, c\rangle \in L(V)).$$

We'll call *c* a *certificate*.

The Language Class NP

An alternative definition for the class NP:

$L \in$ NP iff

- there exists a **deterministic** TM V such that: V is a **verifier** for L, and
- $timereq(V) \in \mathcal{O}(n^k)$ for some k.

"**Deterministic Polynomial-Time Verifying**"

Nondeterministic Deciding and Deterministic Verifying

Theorem 28.9 These two definitions are equivalent:

(1) $L \in$ NP iff there exists a nondeterministic, polynomial-time TM that decides it.

(2) $L \in$ NP iff there exists a deterministic, polynomial-time verifier for it.

Proving That a Language is in NP

- ✓ Exhibit an nondeterministic polynomial-time decider TM to decide it.
- ✓ Exhibit a deterministic polynomial-time verifier TM to verify it.

Languages that are in NP

TSP-DECIDE is in NP

TSP-DECIDE = {<*G*, *cost*> : <*G*> encodes an undirected graph with a positive distance attached to each of its edges and *G* contains a Hamiltonian circuit whose total cost is less than <*cost*>} is in NP.

Theorem 28.10

HAMILTONIAN-CIRCUIT is in NP

HAMILTONIAN-CIRCUIT = {<*G*, *s*> : *G* is an undirected graph and *G* contains a Hamiltonian circuit} is in NP.

Theorem 28.22

CLIQUE is is NP

CLIQUE = {<G, k> : G is an undirected graph with vertices V and edges E, k is an integer, $1 \leq k \leq |V|$, and G contains a k-clique} is in NP.

- A *clique* in G is a subset of V where *every pair of vertices* in the clique is *connected by some edge* in E.
- A k-clique is a clique that contains *exactly k vertices*.

Theorem 28.11

The Satisfiability Problem

SAT = {*w* : *w* is a Boolean wff and *w* is satisfiable} is in NP.

SAT is in NP

SAT = {w : w is a Boolean wff and w is satisfiable} is in NP.

Theorem 28.12

C-SAT: A Restricted Satisfiability Problem

- A *literal* is either a variable or a variable preceded by a single negation symbol.
- A *clause* is either a single literal or the disjunction of two or more literals.
- A wff is in *conjunctive normal form* (or **CNF**) iff it is either a single clause or the conjunction of two or more clauses.
  - Normal form for Boolean expressions

C-SAT is in NP

A Restricted Satisfiability Problem:

C-SAT = { *w* : *w* is a wff in Boolean logic,
w is in conjunctive normal form, &
w is satisfiable} is in NP.

k-SAT: A Restricted Satisfiability Problem

- A wff is in ***k-conjunctive normal form*** (or **k-CNF**) iff it is in conjunctive normal form and each clause contains exactly k literals.

k-SAT is in NP

A Restricted Satisfiability Problem:

k-SAT = { *w* : *w* is a wff in Boolean logic,
w is in k-conjunctive normal form, &
w is satisfiable} is in NP.

3-SAT: A Restricted Satisfiability Problem

- A wff is in ***3-conjunctive normal form*** (or **3-CNF**) iff it is in conjunctive normal form and each clause contains exactly three literals.

3-SAT is in NP

A Restricted Satisfiability Problem:

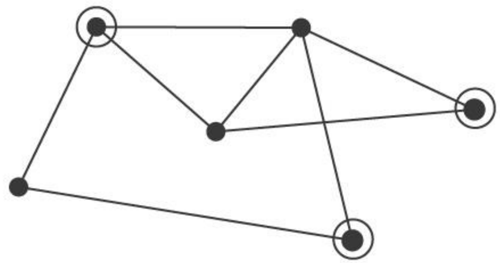
3-SAT = { *w* : *w* is a wff in Boolean logic,
w is in 3-conjunctive normal form, &
w is satisfiable} is in NP.

Theorem 28.13

INDEPENDENT-SET is in NP

INDEPENDENT-SET = {<*G*, *k*> : *G* is an undirected graph and *G* contains an independent set of *at least k* vertices} is in NP.

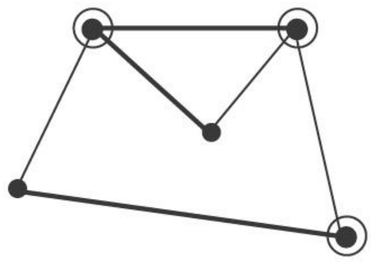
An *independent set* is a set of vertices *no two of which are adjacent.*



VERTEX-COVER is in NP

VERTEX-COVER = {<*G*, *k*>: *G* is an undirected graph and there exists a vertex cover of *G* that contains *at most k* vertices} is in NP.

A *vertex cover* *C* of a graph *G* = (*V*, *E*) is a subset of *V* such that *every edge in E touches at least one of the vertices* in *C*.



SUBSET-SUM is in NP

SUBSET-SUM = {<*S*, *k*> : *S* is a multiset of integers, *k* is an integer, and there exists some subset of *S* whose elements sum to *k*} is in NP.

SET-PARTITION is in NP

SET-PARTITION = {<S> : S is a multiset (i.e., duplicates are allowed) of objects each of which has an associated cost and there exists a way to divide S into two subsets, A and $S – A$, such that the sum of the costs of the elements in A equals the sum of the costs of the elements in $S - A$} is in NP.

KNAPSACK is in NP

KNAPSACK = {<*S*, *v*, *c*> : *S* is a set of objects each of which has an associated cost and an associated value, *v* and *c* are integers, and there exists some way of choosing elements of *S* (duplicates allowed) such that the total cost of the chosen objects is at most *c* and their total value is at least *v*} is in NP.

Notice that, if the cost of each item equals its value, then the KNAPSACK problem becomes the SUBSET-SUM problem.

The Relationship Between P and NP

The P $\subseteq$ NP Question?

P is Contained in NP

Every language in P is also in NP!

Theorem 28.14 P $\subseteq$ NP

Proof Idea:

The Relationship Between P and NP

- Is P = NP? NP $\subseteq$ P?

  No one knows!

The Relationship Between P and NP

- Here are some things we know:

$P \subseteq NP$
$NP \subseteq EXPTIME$
$PSPACE \subseteq EXPTIME$

$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$

$P \neq EXPTIME$ **(Deterministic Time Hirearchy Theorem)**

Polynomial Time Reduction

Mapping Reduction

A ***mapping reduction*** *R* from L_1 to L_2 is a Turing machine that implements some computable function *f* with the property that:

$$\forall x\ (x \in L_1 \leftrightarrow f(x) \in L_2).$$

If $L_1 \leq L_2$ and TM *M* decides L_2, then

$$C(x) = M(R(x)) \text{ will decide } L_1.$$

Deterministic Polynomial Time Mapping Reduction

If $\boldsymbol{R}$ from L_1 to L_2 is **deterministic polynomial time reduction** then

$$L_1 \leq_{\mathbf{P}} L_2$$

- L_1 must be in P if L_2 is: if L_2 is in P then there exists some deterministic, polynomial-time Turing machine M that decides it. So $M(R(x))$ is also a deterministic, polynomial-time Turing machine and it decides L_1.
- L_1 must be in NP if L_2 is: if L_2 is in NP then there exists some nondeterministic, polynomial-time Turing machine M that decides it. So $M(R(x))$ is also a nondeterministic, polynomial-time Turing machine and it decides L_1.

Using Polynomial Time Mapping Reduction in Complexity Proofs

Given L_1 and L_2 and $L_1 \leq_P L_2$, we can use reduction to:

- Prove that L_1 is in P or in NP because we ***already know*** that L_2 is.
- Prove that L_1 would be in P or in NP if we ***could somehow show*** that L_2 is.
  - When we do this, we cluster languages of similar complexity (even if we're not yet sure what that complexity is).
  - In other words, L_1 is no harder than L_2 is.

3-SAT is Reducible to INDEPENDENT-SET

3-SAT = { *w* : *w* is a wff in Boolean logic, *w* is in 3-conjunctive normal form, & *w* is satisfiable }

INDEPENDENT-SET = {<*G*, *k*> : *G* is an undirected graph and *G* contains an independent set of *at least k* vertices}

3-SAT $\leq_P$ INDEPENDENT-SET

Theorem 28.15 3-SAT $\leq_P$ INDEPENDENT-SET

Proof Idea:

A deterministic, polynomial-time reduction **R** from 3-SAT to INDEPENDENT-SET!

3-SAT $\leq_P$ INDEPENDENT-SET

R: A mapping from a Boolean formula in 3-conjunctive normal form to a graph

- Strings in 3-SAT describe formulas that contain literals and clauses.

$$(P \vee Q \vee \neg R) \wedge (R \vee \neg S \vee Q)$$

- Strings in INDEPENDENT-SET describe graphs that contain vertices and edges.

`101/1/11/11/10/10/100/100/101/11/101`

3-SAT $\leq_P$ INDEPENDENT-SET

***R*(<*f*: Boolean formula with *k clauses*>) =**

1. Build a graph *G* by doing the following:

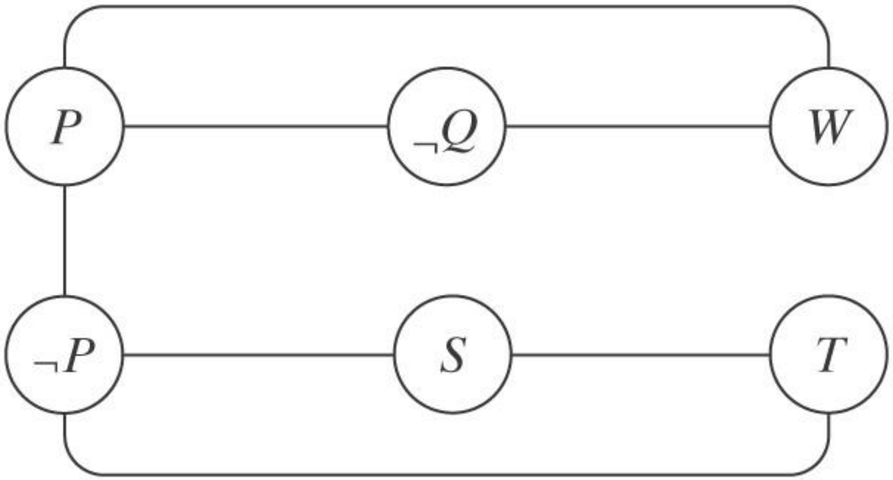
    1.1. Create one vertex for each instance of each literal in *f*.
    1.2. Create an edge between each pair of vertices for symbols in the same clause.
    1.3. Create an edge between each pair of vertices for complementary literals.

2. Return <*G*, *k*>.

3-SAT $\leq_P$ INDEPENDENT-SET

$$(P \vee \neg Q \vee W) \wedge (\neg P \vee S \vee T)$$

$$\Downarrow$$



3-SAT $\leq_P$ INDEPENDENT-SET

- R is a deterministic, polynomial-time reduction.
- Show $<f> \in$ 3-SAT iff $R(<f>) \in$ INDEPENDENT-SET by showing:
  - $<f> \in$ 3-SAT $\rightarrow R(<f>) \in$ INDEPENDENT-SET
  - $R(<f>) \in$ INDEPENDENT-SET $\rightarrow <f> \in$ 3-SAT

3-SAT $\leq_P$ INDEPENDENT-SET

<*f*> $\in$ 3-SAT $\rightarrow$ *R*(<*f*>) $\in$ INDEPENDENT-SET:

<*f*> $\in$ 3-SAT. There is a satisfying assignment *A* to the symbols in *f*. So, *G* contains an independent set *S* of size *k*, built by:

1. From each clause gadget choose one literal that is made positive by *A*.
2. Add the vertex corresponding to that literal to *S*.

S will contain exactly *k* vertices and S is an independent set:

- No two vertices come from the same clause so step 1.2 could not have created an edge between them.
- No two vertices correspond to complimentary literals so step 1.3 could not have created an edge between them.

3-SAT $\leq_P$ INDEPENDENT-SET

R(<*f*>) $\in$ INDEPENDENT-SET $\rightarrow$ <*f*> $\in$ 3-SAT:

R(<*f*>) $\in$ INDEPENDENT-SET. So, the graph *G* that *R* builds contains an independent set *S* of size *k*.

No two vertices in *S* come from the same clause gadget. Since *S* contains at least *k* vertices, no two are from the same clause, and *f* contains *k* clauses, *S* must contain one vertex from each clause.

Build *A* as follows:

1. Assign *True* to each literal that corresponds to a vertex in *S*.
2. Assign arbitrary values to all other literals.

Since each clause will contain at least one literal whose value is *True*, the value of *f* will be *True*.

NP-Complete Problems

NP-Hard and NP-Complete Languages

A language *L* might have these properties:

1. *L* is in NP.
2. Every language in NP is deterministic polynomial-time reducible to *L*.

- *L* is ***NP-hard*** iff it possesses property 2.

  An NP-hard language is at least as hard as any other language in NP.

- *L* is ***NP-complete*** iff it possesses *both* property 1 and property 2.

  All NP-complete languages can be viewed as being equivalently hard.

NP-Hard vs. NP-Complete

SUDOKU = {<*b*>: *b* is a configuration of an $n \times n$ grid and *b* has a solution under the rules of Sudoku}.

- *NP-complete.*

CHESS = {<*b*>: *b* is a configuration of an $n \times n$ chess board and there is a guaranteed win for the current player}.

- *NP-hard, not thought to be in NP.*
- *If fixed number of pieces: PSPACE-complete.*
- *If varialbe number of pieces: EXPTIME-complete.*

SAT: The Satisfiability Problem

The Satisfiability Problem:
Given a Boolean expression, is it satisfiable?

SAT = {*w* : *w* is a wff in Boolean logic and *w* is satisfiable}

SAT is NP-Complete

- ✓ The **first** NP-complete language!
- ✓ **The Cook-Levin Theorem**

The Cook-Levin Theorem

Theorem 28.16 SAT is NP-complete.

Proof Idea:

- ✓ SAT is in NP and
- ✓ SAT is NP-hard.

SAT is in NP

Theorem 28.12 SAT = {w : w is a wff in Boolean logic and w is satisfiable} is in NP.

SAT is NP-Hard

Let L be any language in NP.

Let M be one of the NDTMs that decides L.

Define an algorithm that, given M, constructs a reduction R with the property that:

$$w \in L \text{ iff } R(w) \in \text{SAT}.$$

R takes a string w and returns a Boolean wff that is satisfiable iff $w \in L$.

SAT is NP-Hard

On input w, R uses <M> and constructs a description of the Boolean formula:

$DescribeMonw = Conj_1 \wedge Conj_2 \wedge Conj_3 \wedge Conj_4$.

DescribeMonw will have a satisfying assignment to its variables iff there exists some computational path along which M accepts w.

So, for any NP language L, $L \leq$ SAT.

Then, show that $R(w)$ operates in polynomial time.

Other NP-Complete Languages

Some NP-Complete Languages

- ✓ CSAT
- ✓ 3-SAT

- ✓ INDEPENDENT-SET
- ✓ VETREX-COVER
- ✓ CLIQUE

- ✓ TSP-DECIDE
- ✓ DIRECTED-HAMILTONIAN-CIRCUIT
- ✓ HAMILTONIAN-CIRCUIT

- ✓ SUBSET-SUM
- ✓ SET-PARTITION
- ✓ KNAPSACK
- ✓ SUDOKU

Proving that L is NP-Complete

Proving that L is NP-Complete

Theorem 28.17 If L_1 is NP-complete, $L_1 \leq_P L_2$, and L_2 is in NP, then L_2 is also NP-complete.

Proof Idea:

If L_1 is NP-complete then every other NP language is deterministic, polynomial-time reducible to it.

So let L be any NP language and let R_L be the Turing machine that reduces L to L_1.

If $L_1 \leq_P L_2$, let R_2 be the Turing machine that implements that reduction.

Then L can be deterministic, polynomial-time reduced to L_2 by first applying R_L and then applying R_2.

Since L_2 is in NP and every other language in NP is deterministic, polynomial-time reducible to it, L_2 is NP-complete.

Proving that a New *L* is NP-Complete

NPL_1 NPL_2 NPL_3 NPL_4 NPL...

$\mathbf{L_1}$

$\downarrow$

L

1. *Show that* ***L*** is in NP,
2. Choose $\mathbf{L_1}$ any known NP-complete and *Show that* $\mathbf{L_1} \le_P \mathbf{L}$.

3-SAT: A Restricted Satisfiability Problem

3-SAT = {<*w*> : *w* is a wff in Boolean logic, *w* is in 3-conjunctive normal form and *w* is satisfiable}.

$$(P \vee R \vee \neg T) \wedge (S \vee \neg R \vee W)$$

3-SAT is NP-Complete

Theorem 28.18 3-SAT is NP-complete.

Proof Idea:

- ✓ 3-SAT is in NP.
- ✓ 3-SAT is NP-hard by SAT $\leq_P$ 3-SAT

3-SAT is in NP

Theorem 28.13 3-SAT = {*w* : *w* is a wff in Boolean logic and w is in 3-conjunctive normal form, and *w* is satisfiable} is in NP.

SAT $\leq_P$ 3-SAT

A polynomial-time reduction **R** from SAT to 3-SAT:

***R*(*w*: wff of Boolean logic) =**

1. Use *conjunctiveBoolean* to construct w', where w' is in *conjunctive normal form* and w' is equivalent to w. (Theorem B.1)
2. Use *3-conjunctiveBoolean* to construct w'', where w'' is in *3-conjunctive normal form* and w'' is satisfiable iff w' is. (Theorem B.2)
3. Return w''.

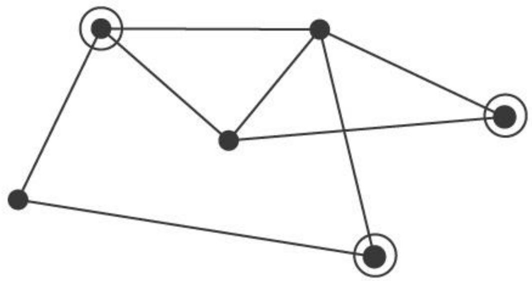
SAT $\leq_P$ 3-SAT

Does *R* run in polynomial time?

1. For *R* to be a reduction from SAT to 3-SAT, it is sufficient to assure that w' is satisifiable iff *w* is.
2. There exists a polynomial-time algorithm that constructs, from any wff *w*, a w' that meets that requirement.
3. If we replace step one of *R* with that algorithm, *R* can be a polynomial-time reduction from SAT to 3-SAT.

INDEPENDENT-SET

INDEPENDENT-SET = {<G, k> : G is an undirected graph and G contains an independent set of *at least k* vertices}.



INDEPENDENT-SET is NP-Complete

Theorem 28.19 INDEPENDENT-SET is NP-complete.

Proof Idea:

✓ INDEPENDENT-SET is in NP and

✓ INDEPENDENT-SET is NP-hard by

3-SAT $\leq_P$ INDEPENDENT-SET

INDEPENDENT-SET is in NP

Proof:

$Ver(<G, k, c>)$ =

1. Check that the number of vertices in c is at least k and no more than $|V|$. If it is not, reject.

2. For each vertex v in c:

   For each edge e in E that has v as one endpoint:

      Check that the other endpoint of e is not in c.

$Timereq(Ver) \in \mathcal{O}(|c| \cdot |E| \cdot |c|)$.

$|c|$ and $|E|$ are polynomial in $|<G, k>|$.

So *Ver* runs in polynomial time.

3-SAT $\leq_P$ INDEPENDENT-SET

3-SAT $\leq_P$ INDEPENDENT-SET

Theorem 28.15 3-SAT $\leq_P$ INDEPENDENT-SET

NP-Complete Problems So Far

SAT

$\leq_P$

3-SAT

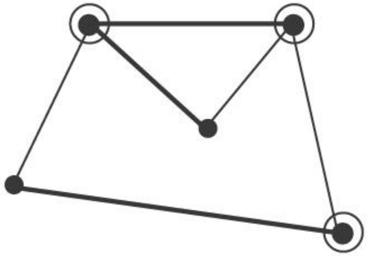
$\leq_P$

INDEPENDENT-SET

VERTEX-COVER

VERTEX-COVER = {<G, k>: G is an undirected graph and there exists a vertex cover of G that contains *at most k* vertices}.

A *vertex cover* C of a graph $G = (V, E)$ is a subset of V such that every edge in E touches at least one of the vertices in C.



VERTEX-COVER is NP-Complete

Theorem 28.20 VERTEX-COVER is NP-complete.

Proof Idea:

✓ VERTEX-COVER is in NP, and

✓ VERTEX-COVER is NP-hard by

$$\text{3-SAT} \leq_{\mathbf{P}} \text{VERTEX-COVER}$$

VERTEX-COVER is in NP

Proof:

Ver(<*G*, *k*, *c*>) =

1. Check that the number of vertices in *c* is at most *min*(*k*, |*V*|). If not, reject.
2. For each vertex *v* in *c* do:
   Find all edges in *E* that have *v* as one endpoint and mark each such edge.
3. Make one final pass through *E* and check whether every edge is marked. If all of them are, accept; otherwise reject.

Timereq(*Ver*) $\in \mathcal{O}(|c| \cdot |E|)$. Both |*c*| and |*E*| are polynomial in |<*G*, *k*>|. So *Ver* runs in polynomial time.

VERTEX-COVER is NP-Hard

3-SAT $\leq_P$ VERTEX-COVER

3-SAT $\leq_P$ VERTEX-COVER

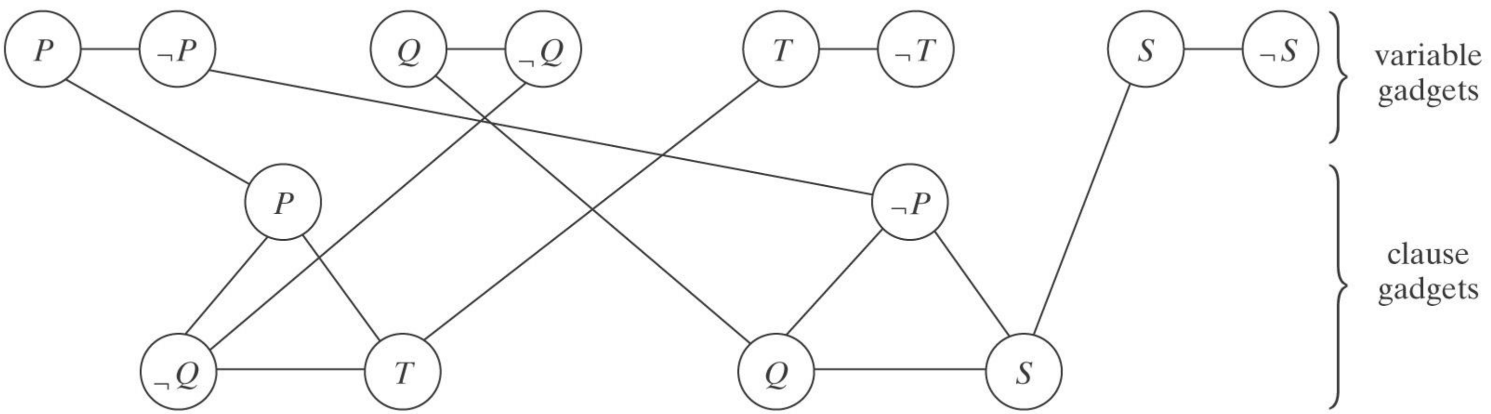
A reduction **R** maps a Boolean formula in 3-conjunctive normal form to a graph:

Given a wff *f*, *R* will exploit two kinds of gadgets:

- A variable gadget: For each variable *x* in *f*, *R* will build a simple graph with two vertices and one edge between them. Label one of the vertices *x* and the other one $\neg x$.
- A clause gadget: For each clause *c* in *f*, *R* will build a graph with three vertices, one for each literal in *c*. There will be an edge between each pair of vertices in this graph.
- Then *R* will build an edge from every vertex in a clause gadget to the vertex of the variable gadget with the same label.

3-SAT $\leq_P$ VERTEX-COVER

$$(P \vee \neg Q \vee T) \wedge (\neg P \vee Q \vee S)$$



3-SAT $\leq_P$ VERTEX-COVER

***R*(<*f*>) =**

1. Build a graph *G* as described above.
2. Let $k = v + 2c$.
3. Return <*G*, *k*>.

- *R* runs in polynomial time.
- To show that it is correct, we must show that:

<*f*> ∈ 3-SAT iff *R*(<*f*>) ∈ VERTEX-COVER

3-SAT $\leq_P$ VERTEX-COVER

$<f> \in$ 3-SAT $\rightarrow R(<f>) \in$ VERTEX-COVER:

There exists a satisfying assignment *A* for *f*.
G contains a vertex cover *C* of size *k*:

1. From each variable gadget, add to *C* the vertex that corresponds to the literal that is true in *A*.
2. Since *A* is a satisfying assignment, there must exist at least one true literal in each clause. Pick one and put the vertices corresponding to the other two into *C*.

C contains exactly *k* vertices and C is a cover of *G*.

3-SAT $\leq_P$ VERTEX-COVER

$R(<f>) \in$ VERTEX-COVER $\rightarrow$ $<f> \in$ 3-SAT:

The graph *G* that *R* builds contains a vertex cover *C* of size *k*.
C must:

- Contain at least one vertex from each variable gadget in order to cover the internal edge in the variable gadget.
- Contain at least two vertices from each clause gadget in order to cover all three internal edges in the clause gadget.

Satisfying those two requirements uses up all $k = v + 2c$ vertices.

3-SAT $\leq_P$ VERTEX-COVER

We can use *C* to show that there exists some satisfying assignment *A* for *f*.

To build *A*,

- assign the value *True* to each literal that is the label for one of the vertices of *C* that comes from a variable gadget.

Then,

A is a satisfying assignment for *f* iff it assigns the value *True* to at least one literal in each of *f*'s clauses.

TSP-DECIDE is NP-Complete

All of these languages are NP-complete:

3-SAT

$\leq_P$

DIRECTED-HAMILTONIAN-CIRCUIT (DHC)

$\leq_P$

HAMILTONIAN-CIRCUIT (HC)

$\leq_P$

TSP-DECIDE

DIRECTED-HAMILTONIAN-CIRCUIT (DHC) is NP-Complete

Theorem 28.21: DIRECTED-HAMILTONIAN-CIRCUIT is in NP-complete.

Proof Idea: By polynomial-time reduction from

3-SAT.

HAMILTONIAN-CIRCUIT (HC) is NP-Complete

Theorem 28.22 HAMILTONIAN-CIRCUIT is in NP-complete.

Proof Idea: By polynomial-time reduction from

DIRECTED-HAMILTONIAN-CIRCUIT.

TSP-DECIDE is NP-Complete

Theorem 28.23 TSP-DECIDE is in NP-complete.

Proof Idea: By polynomial-time reduction from

HAMILTONIAN-CIRCUIT.

P vs. NP-Complete Problems

1. Circuit problems
2. SAT problems
3. Path problems
4. Covering problems
5. Map coloring problems
6. Linear programming problems

1. Two Similar Circuit Problems

- EULERIAN-CIRCUIT, in which we check that there is a circuit that visits every *edge* exactly once, is in P.

- HAMILTONIAN-CIRCUIT, in which we check that there is a circuit that visits every *vertex* exactly once, is NP-complete.

2. Two Similar SAT Problems

- 2-SAT = {<*w*> : *w* is a wff in Boolean logic, *w* is in 2-conjunctive normal form and *w* is satisfiable} is in P.

  $$(\neg P \vee R) \wedge (S \vee \neg T)$$

- 3-SAT = {<*w*> : *w* is a wff in Boolean logic, *w* is in 3-conjunctive normal form and *w* is satisfiable} is NP-complete.

  $$(\neg P \vee R \vee T) \wedge (S \vee \neg T \vee \neg W)$$

3. Two Similar Path Problems

- SHORTEST-PATH = {<*G, u, v, k*>: *G* is an undirected graph, *u* and *v* are vertices in *G, k* ≥ 0, and there exists a path from *u* to *v* whose length is **at most** *k*} is in P.

- LONGEST-PATH = {<*G, u, v, k*>: *G* is an undirected graph, *u* and *v* are vertices in *G, k* ≥ 0, and there exists a path with no repeated edges from *u* to *v* whose length is **at least** *k*} is NP-complete.

4. Two Similar Covering Problems

- EDGE-COVER = {<G, k>: G is an undirected graph and there exists an edge cover of G that contains at most k edges} is in P.

- VERTEX-COVER = {<G, k>: G is an undirected graph and there exists a vertex cover of G that contains at most k vertices} is NP-complete.

5. Two Similar Coloring Problems

- 2-COLORABLE = {<*m*> : *m* can be colored with 2 colors} is in P.
- 3-COLORABLE = {<*m*> : *m* can be colored with 3 colors} is NP-complete.

6. Two Similar Linear Programming Problems

- LINEAR-PROGRAMMING = {<a set of linear inequalities $Ax \leq b$> : there exists a **rational** vector X that satisfies all of the inequalities} is in P.

- INTEGER-PROGRAMMING = {<a set of linear inequalities $Ax \leq b$> : there exists an **integer** vector X that satisfies all of the inequalities} is NP-complete.

The Complexity Class EXPTIME

EXPTIME

= { Problems Solvable in Exponential Time by DTMs }

EXPTIME

For any language *L*,

L $\in$ EXPTIME iff

- there exists some deterministic Turing machine *M* that decides *L* and
- *timereq*(*M*) $\in$ $\mathcal{O}(2^{(n^k)})$ for some positive integer *k*.

CHESS is in EXPTIME

CHESS = {<*b*>: *b* is a configuration of an $n \times n$ chess board and there is a guaranteed win for the current player} is in EXPTIME.

GO

EXPTIME-Completeness

Suppose that:

1. *L* is in EXPTIME.

2. Every language in EXPTIME is deterministic, polynomial-time reducible to *L*.

L is *EXPTIME-hard* iff it possesses property 2.

If it also possesses property 1, it is *EXPTIME-complete*.

CHESS is in EXPTIME-Complete

CHESS = {<*b*>: *b* is a configuration of an $n \times n$ chess board and there is a guaranteed win for the current player}

CHESS is EXPTIME-complete if we add pieces as well as rows and columns.

Time Complexity Hierarchy

$P \subseteq NP \subseteq EXPTIME$

It is not known which of these inclusions is proper.

However, from the Deterministic Time Hierarchy Theorem:

$P \neq EXPTIME$

- It is thought that all of them are proper inclusions.
- A consequence of the fact that $P \neq EXPTIME$ is that we know that there are decidable problems for which no efficient (i.e., polynomial time) decision procedure exists.

Tractability Hierarchy of Decidable Languages

- P
- NP
- EXPTIME

$P \subseteq NP \subseteq EXPTIME$

$P \neq EXPTIME$

$P \subset EXPTIME$

Reading Assignment

Chapter 28:

Sections
28.1
28.2
28.3
28.4
28.5
28.6
28.9

In-Class Exercises

Chapter 28:

1
2
3 - b
8- a
15