NP-Completeness
Outline

P and NP (§13.1)
- Definition of P
- Definition of NP
- Alternate definition of NP

NP-completeness (§13.2)
- Definition of NP-hard and NP-complete
- The Cook-Levin Theorem
More Outline

Some NP-complete problems (§13.3)
- Problem reduction
- SAT (and CNF-SAT and 3SAT)
- Vertex Cover
- Clique
- Hamiltonian Cycle
Back To NP-Completeness
Running Time Revisited

- Input size, $n$
  - To be exact, let $n$ denote the number of bits in a nonunary encoding of the input
- All the polynomial-time algorithms studied so far in this course run in polynomial time using this definition of input size.
  - Exception: any pseudo-polynomial time algorithm
Dealing with Hard Problems

What to do when we find a problem that looks hard...

I couldn’t find a polynomial-time algorithm; I guess I’m too dumb.
Dealing with Hard Problems

Sometimes we can prove a strong lower bound... (but not usually)

I couldn’t find a polynomial-time algorithm, because no such algorithm exists!
Dealing with Hard Problems

NP-completeness let’s us show collectively that a problem is hard.

I couldn’t find a polynomial-time algorithm, but neither could all these other smart people.
Polynomial-Time Decision Problems

To simplify the notion of “hardness,” we will focus on the following:

- Polynomial-time as the cut-off for efficiency
- Decision problems: output is 1 or 0 (“yes” or “no”)
  - Examples:
    - Is a given circuit satisfiable?
    - Does a text T contain a pattern P?
    - Does an instance of 0/1 Knapsack have a solution with benefit at least K?
    - Does a graph G have an MST with weight at most K?
Problems and Languages

A language \( L \) is a set of strings defined over some alphabet \( \Sigma \)
- Don’t be fooled here: the key is that your “language” will be a binary encoding of a specific instance of a specific problem.

Every decision algorithm \( A \) defines a language \( L \)
- \( L \) is the set consisting of every string \( x \) such that \( A \) outputs “yes” on input \( x \).
- We say “\( A \) accepts \( x \)” in this case
  - Example: Euler tours.
  - An Euler tour of a directed graph is a cycle that visits all of the vertices in the graph only once (except for the initial vertex, which obviously is visited twice).
  - If \( A \) determines whether or not a given graph \( G \) has an Euler tour, then the language \( L \) for \( A \) is all graphs with Euler tours.
Problems and Languages

Note that saying an algorithm $A$ defines a language $L$ means that a string $x$ is in $L$ if and only if $A$ outputs "yes" when fed string $x$. 
A **complexity class** is a collection of languages

- P is the complexity class consisting of all languages that are accepted by **polynomial-time** algorithms

- This is a funny way of saying things, but necessary for rigor.

- Ex. If question is ``Does this graph have an Euler tour?``, and there is a polynomial time algorithm A that will say ``yes`` for all graphs that have Euler tours, then the language defined by A (which will include all strings representing graphs with Euler tours) is in P.

  - For those of us who feel comfortable with natural language: the problem of determining whether a graph has an Euler tour would be solvable in polynomial time.
A *complexity class* is a collection of languages. P is the complexity class consisting of all languages that are accepted by *polynomial-time* algorithms.

For each language L in P there is a polynomial-time decision algorithm A for L.

- If $n = |x|$, for $x$ in L, then A runs in $p(n)$ time on input x.
- The function $p(n)$ is some polynomial.
The Complexity Class NP

- We say that an algorithm is non-deterministic if it uses the following operation:
  - Choose(b): chooses a bit b
  - Can be used to choose an entire string y (with |y| choices)

- We say that a non-deterministic algorithm A accepts a string x if there exists some sequence of choose operations that causes A to output “yes” on input x.

- NP is the complexity class consisting of all languages accepted by polynomial-time non-deterministic algorithms.
NP example

Problem: Decide if a graph has an MST of weight $K$ or less

Algorithm:
1. Non-deterministically choose a set $T$ of $n-1$ edges
2. Test that $T$ forms a spanning tree
3. Test that $T$ has weight at most $K$

Analysis: Testing takes $O(n+m)$ time, so this algorithm runs in polynomial time.
The Complexity Class NP

Alternate Definition

- We say that an algorithm B verifies the acceptance of a language L if and only if, for any x in L, there exists a certificate y such that B outputs “yes” on input (x,y).
  - The certificate might be, for example, the path that describes the Euler tour (if this is the problem being considered).
- NP is the complexity class consisting of all languages verified by polynomial-time algorithms.

- We know: P is a subset of NP.
- Major open question: P=NP?
- Most researchers believe that P and NP are different.
NP example (2)

Problem: Decide if a graph has an MST of weight $K$

Verification Algorithm:
1. Use as a certificate, $y$, a set $T$ of n-1 edges
2. Test that $T$ forms a spanning tree
3. Test that $T$ has weight at most $K$

Analysis: Verification takes $O(n+m)$ time, so this algorithm runs in polynomial time.
Equivalence of the Two Definitions

Suppose A is a non-deterministic algorithm
- Let y be a certificate consisting of all the outcomes of the choose steps that A uses
- We can create a verification algorithm that uses y instead of A’s choose steps
- If A accepts on x, then there is a certificate y that allows us to verify this (namely, the choose steps A made)
- If A runs in polynomial-time, so does this verification algorithm

Suppose B is a verification algorithm
- Non-deterministically choose a certificate y
- Run B on y
- If B runs in polynomial-time, so does this non-deterministic algorithm
An Interesting Problem

A Boolean circuit is a circuit of AND, OR, and NOT gates; the CIRCUIT-SAT problem is to determine if there is an assignment of 0’s and 1’s to a circuit’s inputs so that the circuit outputs 1.
CIRCUIT-SAT is in NP

Non-deterministically choose a set of inputs and the outcome of every gate, then test each gate’s I/O.

Logic Gates:

- NOT
- OR
- AND

Inputs:
- 0
- 1

Output:
- 1
NP-Completeness

A problem (language) \( L \) is **NP-hard** if every problem in NP can be reduced to \( L \) in polynomial time.

That is, for each language \( M \) in NP, we can take an input \( x \) for \( M \), **transform** it in polynomial time to an input \( x' \) for \( L \) such that \( x \) is in \( M \) if and only if \( x' \) is in \( L \).

\( L \) is **NP-complete** if it’s in NP and is NP-hard.
Cook-Levin Theorem

- CIRCUIT-SAT is NP-complete.
  - We already showed it is in NP.

To prove it is NP-hard, we have to show that every language in NP can be reduced to it.
  - Let $M$ be in NP, and let $x$ be an input for $M$.
  - Let $y$ be a certificate that allows us to verify membership in $M$ in polynomial time, $p(n)$, by some algorithm $D$.
  - Let $S$ be a circuit of size at most $O(p(n)^2)$ that simulates a computer (details omitted...)

\[ \text{NP} \xrightarrow{\text{poly-time}} M \xrightarrow{\text{poly-time}} \text{CIRCUIT-SAT} \]
Cook-Levin Proof

We can build a circuit that simulates the verification of $x$’s membership in $M$ using $y$.

- Let $W$ be the working storage for $D$ (including registers, such as program counter); let $D$ be given in RAM “machine code.”
- Simulate $p(n)$ steps of $D$ by replicating circuit $S$ for each step of $D$. Only input: $y$.
- Circuit is satisfiable if and only if $x$ is accepted by $D$ with some certificate $y$.
- Total size is still polynomial: $O(p(n)^3)$.
Some Thoughts about P and NP

Belief: P is a proper subset of NP.
Implication: the NP-complete problems are the hardest in NP.
Why: Because if we could solve an NP-complete problem in polynomial time, we could solve every problem in NP in polynomial time.
That is, if an NP-complete problem is solvable in polynomial time, then P=NP.
Since so many people have attempted without success to find polynomial-time solutions to NP-complete problems, showing your problem is NP-complete is equivalent to showing that a lot of smart people have worked on your problem and found no polynomial-time algorithm.
Showing NP-Completeness
Problem Reduction

- A language $M$ is polynomial-time reducible to a language $L$ if an instance $x$ for $M$ can be transformed in polynomial time to an instance $x'$ for $L$ such that $x$ is in $M$ if and only if $x'$ is in $L$.
  - Denote this by $M \rightarrow L$.

- A problem (language) $L$ is **NP-hard** if every problem in NP is polynomial-time reducible to $L$.

- A problem (language) is **NP-complete** if it is in NP and it is NP-hard.

- CIRCUIT-SAT is NP-complete:
  - CIRCUIT-SAT is in NP
  - For every $M$ in NP, $M \rightarrow$ CIRCUIT-SAT.
Problem Reduction

- A general problem \( M \) is polynomial-time reducible to a general problem \( L \) if an instance \( x \) of problem \( M \) can be transformed in polynomial time to an instance \( x' \) of problem \( L \) such that the solution to \( x \) is yes if and only if the solution to \( x' \) is yes.
  - Denote this by \( M \rightarrow L \).
- A problem (language) \( L \) is NP-hard if every problem in NP is polynomial-time reducible to \( L \).
- A problem (language) is NP-complete if it is in NP and it is NP-hard.
- CIRCUIT-SAT is NP-complete:
  - CIRCUIT-SAT is in NP
  - For every \( M \) in NP, \( M \rightarrow \) CIRCUIT-SAT.
Transitivity of Reducibility

- If A → B and B → C, then A → C.
  - An input x for A can be converted to x’ for B, such that x is in A if and only if x’ is in B. Likewise, for B to C.
  - Convert x’ into x’’ for C such that x’ is in B iff x’’ is in C.
  - Hence, if x is in A, x’ is in B, and x’’ is in C.
  - Likewise, if x’’ is in C, x’ is in B, and x is in A.
  - Thus, A → C, since polynomials are closed under composition.

- Types of reductions:
  - **Local replacement:** Show A → B by dividing an input to A into components and show how each component can be converted to a component for B.
  - **Component design:** Show A → B by building special components for an input of B that enforce properties needed for A, such as “choice” or “evaluate.”
A Boolean formula is a formula where the variables and operations are Boolean (0/1):

- \((a+b+\neg d+e)(\neg a+\neg c)(\neg b+c+d+e)(a+\neg c+\neg e)\)
- OR: +, AND: (times), NOT: \(\neg\)

SAT: Given a Boolean formula \(S\), is \(S\) satisfiable, that is, can we assign 0’s and 1’s to the variables so that \(S\) is 1 (“true”)?

- Easy to see that CNF-SAT is in NP:
  - Non-deterministically choose an assignment of 0’s and 1’s to the variables and then evaluate each clause. If they are all 1 (“true”), then the formula is satisfiable.
CNF-SAT is NP-complete

- Reduce CIRCUIT-SAT to CNF-SAT.
  - Given a Boolean circuit, make a variable for every input and gate.
  - Create a sub-formula for each gate, characterizing its effect. Form the formula as the output variable AND-ed with all these sub-formulas:
    - Example: $m((a+b)\leftrightarrow e)(c\leftrightarrow f)(d\leftrightarrow g)(e\leftrightarrow h)(ef\leftrightarrow i)\ldots$

The formula is satisfiable if and only if the Boolean circuit is satisfiable.
3SAT

- The SAT problem is still NP-complete even if the formula is a conjunction of disjuncts, that is, it is in conjunctive normal form (CNF).
- The SAT problem is still NP-complete even if it is in CNF and every clause has just 3 literals (a variable or its negation):
  - \((a+b+\neg d)(\neg a+\neg c+e)(\neg b+d+e)(a+\neg c+\neg e)\)
- Reduction from SAT (See §13.3.1).
Vertex Cover

- A vertex cover of graph $G=(V,E)$ is a subset $W$ of $V$, such that, for every edge $(a,b)$ in $E$, $a$ is in $W$ or $b$ is in $W$.

- VERTEX-COVER: Given a graph $G$ and an integer $K$, does $G$ have a vertex cover of size at most $K$?

- VERTEX-COVER is in NP: Non-deterministically choose a subset $W$ of size $K$ and check that every edge is covered by $W$. 
Vertex-Cover is NP-complete

Reduce 3SAT to VERTEX-COVER.

Let $S$ be a Boolean formula in CNF with each clause having 3 literals.

For each variable $x$, create a node for $x$ and $\neg x$, and connect these two:

For each clause $C_i = (a+b+c)$, create a triangle and connect the three nodes.
Vertex-Cover is NP-complete

Completing the construction

- Connect each literal in a clause triangle to its copy in a variable pair.
- E.g., for a clause $C_i = (\neg x + y + z)$

Let $n = \#$ of variables
Let $m = \#$ of clauses
Set $K = n + 2m$
Vertex-Cover is NP-complete

- Example: \((a+b+c)(\neg a+b+\neg c)(\neg b+\neg c+\neg d)\)
- Graph has vertex cover of size \(K=4+6=10\) iff formula is satisfiable.
Why? (satisfiable $\Rightarrow$ cover)

- Suppose there is an assignment of Boolean values that satisfies $S$
- Build a subset of vertices that contains each literal that is assigned 1 by satisfying assignment
- For each clause, the satisfying assignment must assign one to at least one of the summands. Include the other two vertices in the vertex cover.
- The cover has size $n + 2m$ (as required).
Is What We Described a Cover?

- Each edge in a truth setting component is covered.
- Each edge in a clause satisfying component is covered.
- Two of three edges incident on a clause satisfying component is covered.
- An edge (incident to a clause satisfying component) not covered by a vertex in the component must be covered by a node in C labeled with a literal, since the corresponding literal is 1 (by how we chose the vertices to be covered in the clause satisfying components).
Why? (cover \(\Rightarrow\) satisfiable)

- Suppose there is a cover \(C\) with size at most \(n + 2m\)
- It must contain at least one vertex from each truth setting component, and two from each clause satisfying component, so size is at least \(n + 2m\) (so exactly that)
- So, one edge incident to any clause satisfying component is not covered by a vertex in the clause satisfying component. This edge must be covered by the other endpoint, which is labeled with a literal.
Why? (cover $\Rightarrow$ satisfiable)

- We can associate the literal associated with this node 1 and each clause in $S$ is satisfied, hence $S$ is satisfied.
- Bottom line: $S$ is satisfiable iff $G$ has a vertex cover of size at most $n + 2m$.
- Bottom line 2: Vertex Cover is NP-Complete
Clique

- A **clique** of a graph $G=(V,E)$ is a subgraph $C$ that is fully-connected (every pair in $C$ has an edge).

- **CLIQUE**: Given a graph $G$ and an integer $K$, is there a clique in $G$ of size at least $K$?

  - **CLIQUE** is in **NP**: non-deterministically choose a subset $C$ of size $K$ and check that every pair in $C$ has an edge in $G$.

This graph has a clique of size 5
CLIQUE is NP-Complete

- Reduction from VERTEX-COVER.
- A graph $G$ has a vertex cover of size at most $K$ if and only if its complement has a clique of size at least $n-K$. 

![Graph G](image1)

![Graph G'](image2)
Some Other NP-Complete Problems

**SET-COVER:** Given a collection of $m$ sets, are there $K$ of these sets whose union is the same as the whole collection of $m$ sets?

- NP-complete by reduction from VERTEX-COVER

**SUBSET-SUM:** Given a set of integers and a distinguished integer $K$, is there a subset of the integers that sums to $K$?

- NP-complete by reduction from VERTEX-COVER
Some Other NP-Complete Problems

- **0/1 Knapsack**: Given a collection of items with weights and benefits, is there a subset of weight at most $W$ and benefit at least $K$?
  - NP-complete by reduction from SUBSET-SUM

- **Hamiltonian-Cycle**: Given an graph $G$, is there a cycle in $G$ that visits each vertex exactly once?
  - NP-complete by reduction from VERTEX-COVER

- **Traveling Saleperson Tour**: Given a complete weighted graph $G$, is there a cycle that visits each vertex and has total cost at most $K$?
  - NP-complete by reduction from Hamiltonian-Cycle.
Approximation Algorithms
Outline and Reading

Approximation Algorithms for NP-Complete Problems (§13.4)
  - Approximation ratios
  - Polynomial-Time Approximation Schemes (§13.4.1)
    - 2-Approximation for Vertex Cover (§13.4.2)
    - 2-Approximation for TSP special case (§13.4.3)
  - Log $n$-Approximation for Set Cover (§13.4.4)
Approximation Ratios

Optimization Problems

- We have some problem instance $x$ that has many feasible “solutions”.
- We are trying to minimize (or maximize) some cost function $c(S)$ for a “solution” $S$ to $x$. For example,
  - Finding a minimum spanning tree of a graph
  - Finding a smallest vertex cover of a graph
  - Finding a smallest traveling salesman tour in a graph
Approximation Ratios

- An approximation produces a solution $T$
  - $T$ is a $k$-approximation to the optimal solution $OPT$ if $c(T)/c(OPT) \leq k$ (assuming a min. prob.; a maximization approximation would be the reverse)
A problem $L$ has a **polynomial-time approximation scheme (PTAS)** if it has a polynomial-time $(1+\varepsilon)$-approximation algorithm, for any fixed $\varepsilon > 0$ (this value can appear in the running time).

$0/1$ Knapsack has a PTAS, with a running time that is $O(n^3/\varepsilon)$. Please see §13.4.1 in Goodrich-Tamassia for details.
Vertex Cover

- A **vertex cover** of graph $G=(V,E)$ is a subset $W$ of $V$, such that, for every $(a,b)$ in $E$, $a$ is in $W$ or $b$ is in $W$.

- **OPT-VERTEX-COVER**: Given a graph $G$, find a vertex cover of $G$ with smallest size.

- **OPT-VERTEX-COVER** is NP-hard.
A 2-Approximation for Vertex Cover

- Every chosen edge $e$ has both ends in $C$
- But $e$ must be covered by an optimal cover; hence, one end of $e$ must be in OPT
- Thus, there is at most twice as many vertices in $C$ as in OPT.
- That is, $C$ is a 2-approx. of OPT
- Running time: $O(m)$

Algorithm $\text{VertexCoverApprox}(G)$

- **Input** graph $G$
- **Output** a vertex cover $C$ for $G$

1. $C \leftarrow$ empty set
2. $H \leftarrow G$
3. while $H$ has edges
   - $e \leftarrow H.\text{removeEdge}(H.\text{anEdge}())$
   - $v \leftarrow H.\text{origin}(e)$
   - $w \leftarrow H.\text{destination}(e)$
   - $C.\text{add}(v)$
   - $C.\text{add}(w)$
   - for each $f$ incident to $v$ or $w$
     - $H.\text{removeEdge}(f)$
4. return $C$
**Special Case of the Traveling Salesperson Problem**

**OPT-TSP:** Given a complete, weighted graph, find a cycle of minimum cost that visits each vertex.

- **OPT-TSP** is NP-hard
- Special case: edge weights satisfy the triangle inequality (which is common in many applications):
  
  \[ w(a,b) + w(b,c) \geq w(a,c) \]

![Diagram of a triangle with edge weights 5, 4, and 7]
A 2-Approximation for TSP
Special Case

Algorithm $\text{TSPApprox}(G)$

- **Input** weighted complete graph $G$, satisfying the triangle inequality
- **Output** a TSP tour $T$ for $G$

1. $M \leftarrow$ a minimum spanning tree for $G$
2. $P \leftarrow$ an Euler tour traversal of $M$, starting at some vertex $s$
3. $T \leftarrow$ empty list
4. for each vertex $v$ in $P$ (in traversal order)
   - if this is $v$’s first appearance in $P$ then
     - $T$.insertLast($v$)
   - $T$.insertLast($s$)
5. return $T$
A 2-Approximation for TSP

Special Case - Proof

- The optimal tour is a spanning tour; hence |M|≤|OPT|.
- The Euler tour \( P \) visits each edge of \( M \) twice; hence |P|=2|M|
- Each time we shortcut a vertex in the Euler Tour we will not increase the total length, by the triangle inequality \( w(a,b) + w(b,c) > w(a,c) \); hence, |T|≤|P|.
- Therefore, |T|≤|P|=2|M|≤2|OPT|

Output tour \( T \)
(at most the cost of \( P \))

Euler tour \( P \) of MST \( M \)
(twice the cost of \( M \))

Optimal tour \( OPT \)
(at least the cost of MST \( M \))
Set Cover

- **OPT-SET-COVER**: Given a collection of \( m \) sets, find the smallest number of them whose union is the same as the whole collection of \( m \) sets?
  - OPT-SET-COVER is NP-hard

- Greedy approach produces an \( O(\log n) \)-approximation algorithm. See §13.4.4 for details.

---

**Algorithm** \( \text{SetCoverApprox}(G) \)

**Input** a collection of sets \( S_1 \ldots S_m \)

**Output** a subcollection \( C \) with same union

\[
F \leftarrow \{S_1, S_2, \ldots, S_m\}
\]

\( C \leftarrow \) empty set

\( U \leftarrow \) union of \( S_1 \ldots S_m \)

while \( U \) is not empty

\[
S_i \leftarrow \text{set in } F \text{ with most elements in } U
\]

\( F.\text{remove}(S_i) \)

\( C.\text{add}(S_i) \)

Remove all elements in \( S_i \) from \( U \)

return \( C \)