Polynomial-time computations

- An algorithm has time complexity $O(T(n))$ if it runs in time at most $cT(n)$ for <u>every</u> input of length n.
- An algorithm is a polynomial-time algorithm if its time complexity is $O(p(n))$, where $p(n)$ is polynomial in n.

Polynomials

- If $f(n)$ = polynomial function in n,

    then $f(n) = O(n^c)$, for some fixed constant c
- If $f(n)$ = exponential (super-polynomial) function in n,

    then $f(n) = \omega(n^c)$, for any constant c
- Composition of polynomial functions are also polynomial, i.e.,

    $f(g(n))$ = polynomial if $f()$ and $g()$ are polynomial
- If an algorithm calls another polynomial-time subroutine a polynomial number of times, then the time complexity is polynomial.

The class $\mathcal{P}$

- A problem is in $\mathcal{P}$ if there exists a polynomial-time algorithm that solves the problem.
- Examples of $\mathcal{P}$
  - ***DFS:*** Linear-time algorithm exists
  - ***Sorting:*** $O(n \log n)$-time algorithm exists
  - ***Bubble Sort:*** Quadratic-time algorithm $O(n^2)$
  - ***APSP:*** Cubic-time algorithm $O(n^3)$
- $\mathcal{P}$ is therefore a class of problems (not algorithms)!

The class $\mathcal{NP}$

- A problem is in $\mathcal{NP}$ if there exists a non-deterministic polynomial-time algorithm that solves the problem.
- A problem is in $\mathcal{NP}$ if there exists a (deterministic) polynomial-time algorithm that verifies a solution to the problem.
- All problems in $\mathcal{P}$ are in $\mathcal{NP}$

TSP: Traveling Salesperson Problem

- **Input:**
  - Weighted graph, G
  - Length bound, B
- **Output:**
  - Is there a traveling salesperson tour in G of length at most B?
- Is TSP in $\mathcal{NP}$?
  - YES. Easy to verify a given solution.
- Is TSP in $\mathcal{P}$?
  - OPEN!
  - One of the greatest unsolved problems of this century!
  - Same as asking: **Is $\mathcal{P} = \mathcal{NP}$?**

So, what is *NP-Complete*?

- *NP-Complete* problems are the "hardest" problems in *NP*.
- We need to formalize the notion of "hardest".

Terminology

- Problem:
  - An abstract problem is a function (relation) from a set I of instances of the problem to a set S of solutions.

    $$p: I \rightarrow S$$
  - An instance of a problem p is obtained by assigning values to the parameters of the abstract problem.
  - Thus, describing the set of all instances (I.e., possible inputs) and the set of corresponding outputs defines a problem.
- Algorithm:
  - An algorithm that solves problem p must give correct solutions to all instances of the problem.
- Polynomial-time algorithm:

Terminology (Cont'd)

- Input Length:
  - length of an <u>encoding</u> of an instance of the problem.
  - Time and space complexities are written in terms of it.
- Worst-case time/space complexity of an algorithm
  - Is the maximum time/space required by the algorithm on any input of length n.
- Worst-case time/space complexity of a problem
  - UPPER BOUND: worst-case time complexity of best existing algorithm that solves the problem.
  - LOWER BOUND: (provable) worst-case time complexity of best algorithm (need not exist) that could solve the problem.
  - LOWER BOUND $\leq$ UPPER BOUND
- Complexity Class $\mathcal{P}$:
  - Set of all problems *p* for which polynomial-time algorithms exist

Terminology (Cont'd)

- Decision Problems:
  - Are problems for which the solution set is {yes, no}
  - Example: Does a given graph have an odd cycle?
  - Example: Does a given weighted graph have a TSP tour of length at most B?
- Complement of a decision problem:
  - Are problems for which the solution is "complemented".
  - Example: Does a given graph NOT have an odd cycle?
  - Example: Is every TSP tour of a given weighted graph of length greater than B?
- Optimization Problems:
  - Are problems where one is maximizing (or minimizing) some objective function.
  - Example: Given a weighted graph, find a MST.
  - Example: Given a weighted graph, find an optimal TSP tour.
- Verification Algorithms:
  - Given a problem instance i and a certificate s, is s a solution for instance i?

Terminology (Cont'd)

- Complexity Class $\mathcal{P}$:
  - Set of all problems *p* for which polynomial-time algorithms exist.
- Complexity Class $\mathcal{NP}$:
  - Set of all problems *p* for which polynomial-time verification algorithms exist.
- Complexity Class *co*-$\mathcal{NP}$:
  - Set of all problems *p* for which polynomial-time verification algorithms exist for their **complements**, I.e., their complements are in $\mathcal{NP}$.

Terminology (Cont'd)

- **Reductions:** $p_1 \rightarrow p_2$
  - A problem p_1 is reducible to p_2, if there exists an algorithm R that takes an instance i_1 of p_1 and outputs an instance i_2 of p_2, with the constraint that the solution for i_1 is YES if and only if the solution for i_2 is YES.
  - Thus, R converts YES (NO) instances of p_1 to YES (NO) instances of p_2.
- Polynomial-time reductions: $p_1 \xrightarrow{P} p_2$
  - Reductions that run in polynomial time.

- If $p_1 \xrightarrow{P} p_2$, then
  - If p_2 is easy, then so is p_1. $p_2 \in \mathcal{P} \Rightarrow p_1 \in \mathcal{P}$
  - If p_1 is hard, then so is p_2. $p_1 \notin \mathcal{P} \Rightarrow p_2 \notin \mathcal{P}$

What are *NP-Complete* problems?

- These are the hardest problems in *NP*.
- A problem p is *NP-Complete* if
  - there is a polynomial-time reduction from **every** problem in *NP* to p.
  - $p \in NP$
- How to prove that a problem is *NP-Complete*?

- **Cook's Theorem**: [1972]
  - The SAT problem is *NP-Complete*.

Steve Cook, Richard Karp, Leonid Levin

NP-Complete vs *NP-Hard*

- A problem p is *NP-Complete* if
  - there is a polynomial-time reduction from **<u>every</u>** problem in *NP* to p.
  - $p \in NP$
- A problem p is *NP-Hard* if
  - there is a polynomial-time reduction from **<u>every</u>** problem in *NP* to p.

The SAT Problem: an example

- Consider the boolean expression:

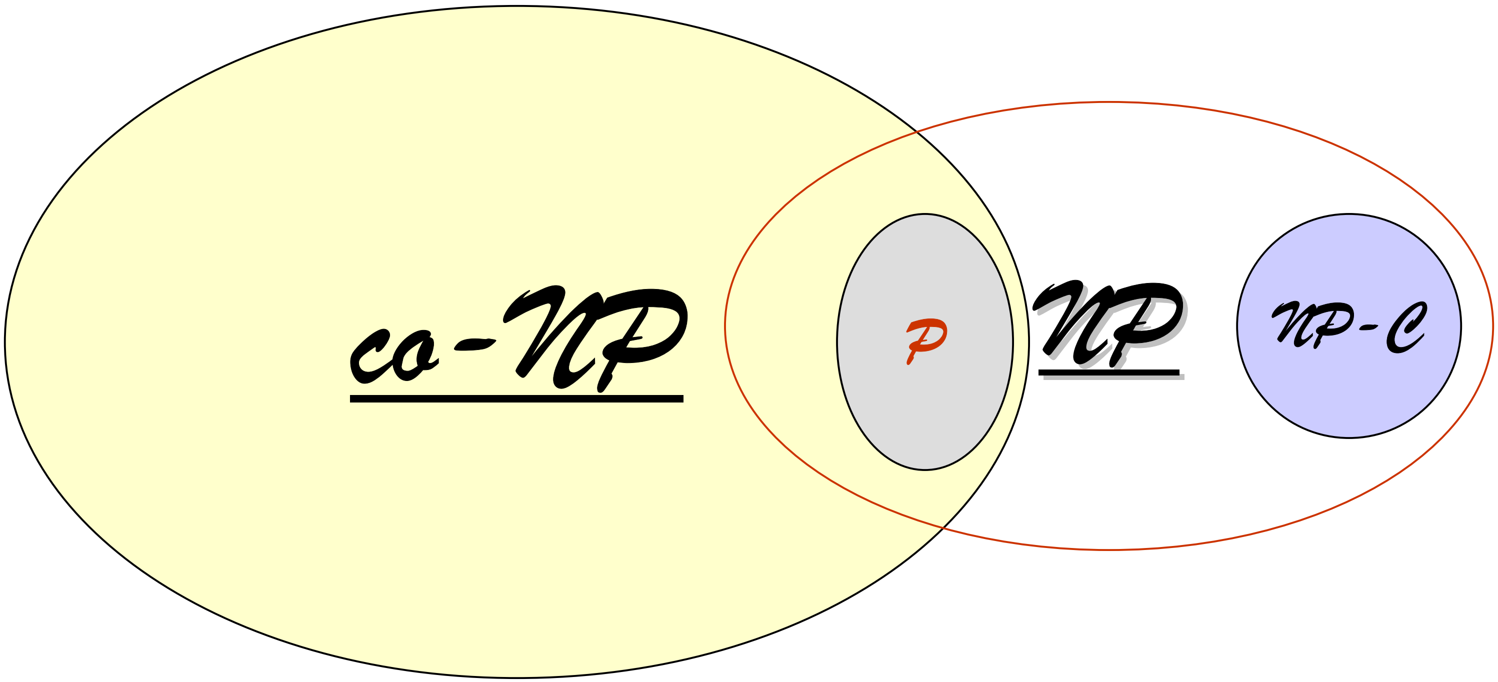
  $C = (a \vee \neg b \vee c) \wedge (\neg a \vee d \vee \neg e) \wedge (a \vee \neg d \vee \neg c)$
- Is C satisfiable?
- Does there exist a True/False assignments to the boolean variables a, b, c, d, e, such that C is True?
- Set a = True and d = True. The others can be set arbitrarily, and C will be true.
- If C has 40,000 variables and 4 million clauses, then it becomes hard to test this.
- If there are n boolean variables, then there are 2^n different truth value assignments.
- However, a solution can be quickly verified!

The SAT (Satisfiability) Problem

- Input: Boolean expression C in Conjunctive normal form (CNF) in n variables and m clauses.
- Question: Is C satisfiable?
  - Let $C = C_1 \wedge C_2 \wedge \ldots \wedge C_m$
  - Where each $C_i = \left(y_1^i \vee y_2^i \vee \cdots \vee y_{k_i}^i\right)$
  - And each $y_j^i \in \{x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n\}$
  - We want to know if there exists a truth assignment to all the variables in the boolean expression C that makes it true.
- Steve Cook showed that the problem of deciding whether a non-deterministic Turing machine T accepts an input w or not can be written as a boolean expression C_T for a SAT problem. The boolean expression will have length bounded by a polynomial in the size of T and w.

- How to now prove Cook's theorem? Is SAT in $\mathcal{NP}$?
- Can every problem in $\mathcal{NP}$ be poly. reduced to it ?

The problem classes and their relationships



More *NP-Complete* problems

3SAT

- **Input**: Boolean expression C in Conjunctive normal form (CNF) in n variables and m clauses. Each clause has at most three literals.
- **Question**: Is C satisfiable?
  - Let $C = C_1 \wedge C_2 \wedge \ \dots \ \wedge C_m$
  - Where each $C_i = \left(y_1^i \vee y_2^i \vee y_3^i\right)$
  - And each $y_j^i \in \{x_1, \neg x_1, x_2, \neg x_2, \dots, x_n, \neg x_n\}$
  - We want to know if there exists a truth assignment to all the variables in the boolean expression C that makes it true.

3SAT is *NP-Complete.*

More *NP-Complete* problems?

2SAT

- Input: Boolean expression C in Conjunctive normal form (CNF) in n variables and m clauses. Each clause has at most <u>three</u> literals.
- Question: Is C satisfiable?
  - Let $C = C_1 \wedge C_2 \wedge \ldots \wedge C_m$
  - Where each $C_i = \left(y_1^i \vee y_2^i\right)$
  - And each $y_j^i \in \{x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n\}$
  - We want to know if there exists a truth assignment to all the variables in the boolean expression C that makes it true.

2SAT is in $\mathcal{P}$.

3SAT is *NP-Complete*

- 3SAT is in *NP*.
- SAT can be reduced in polynomial time to 3SAT.
- This implies that every problem in *NP* can be reduced in polynomial time to 3SAT. Therefore, 3SAT is *NP-Complete*.
- So, we have to design an algorithm such that:
- Input: an instance C of SAT
- Output: an instance C' of 3SAT such that satisfiability is retained. In other words, C is satisfiable if and only if C' is satisfiable.

3SAT is *NP-Complete*

- Let C be an instance of SAT with clauses $C_1, C_2, \ldots, C_m$
- Let C_i be a disjunction of $k > 3$ literals.

  $C_i = \qquad y_1 \vee y_2 \vee \ldots \vee y_k$
- Rewrite C_i as follows:

$$
\begin{aligned}
C'_i = \quad & (y_1 \vee y_2 \vee z_1) \wedge \\
& (\neg z_1 \vee y_3 \vee z_2) \wedge \\
& (\neg z_2 \vee y_4 \vee z_3) \wedge \\
& \ldots \\
& (\neg z_{k-3} \vee y_{k-1} \vee y_k)
\end{aligned}
$$

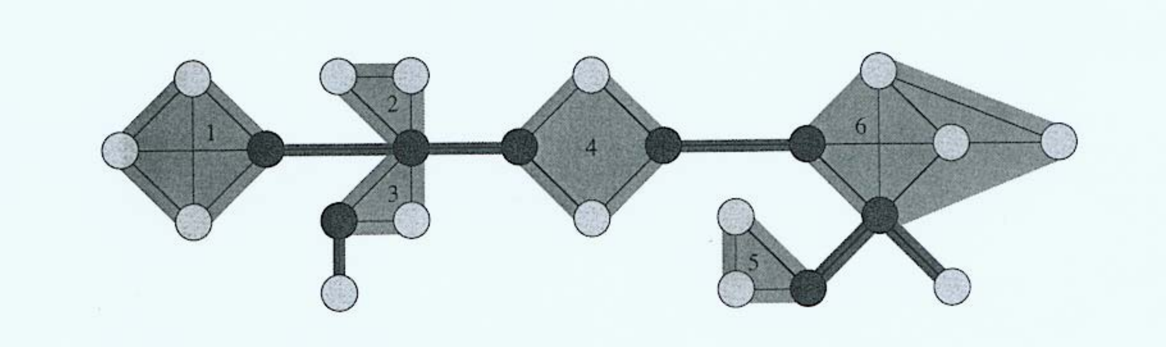
- Claim: C_i is satisfiable if and only if C'_i is satisfiable.

2SAT is in $\mathcal{P}$

- If there is only one literal in a clause, it must be set to true.
- If there are two literals in some clause, and if one of them is set to false, then the other must be set to true.
- Using these constraints, it is possible to check if there is some inconsistency.
- How? Homework problem!

The CLIQUE Problem

- A **clique** is a completely connected subgraph.

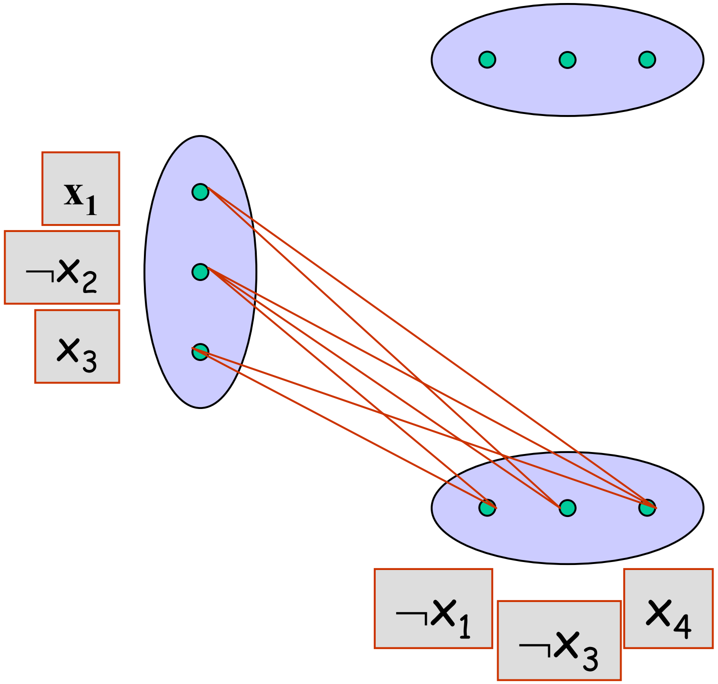


CLIQUE

- Input: Graph G(V,E) and integer k
- Question: Does G have a clique of size k?

CLIQUE is *NP-Complete*

- CLIQUE is in *NP*.
- Reduce 3SAT to CLIQUE in polynomial time.
- $F = (x_1 \vee \neg x_2 \vee x_3)\ (\neg x_1 \vee \neg x_3 \vee x_4)\ (x_2 \vee x_3 \vee \neg x_4)\ (\neg x_1 \vee \neg x_2 \vee x_3)$



F is satisfiable if and only if G has a clique of size k where k is the number of clauses in F.