Chapter 7
Time Complexity

Overview

- previously, we looked at whether a problem was solvable
- even if it is solvable, however, it may not be solvable practically due to time or memory
- computational complexity theory
  - considers time, memory, and other resources required for solving computational problems
  - we will focus on time

Measuring Complexity

- example: $A = \{0^k1^k \mid k \geq 0\}$
  - A is context-free, so it is decidable
  - how much time does a single-tape TM need to decide A?
  - low-level TM description with head motion so that we can count the number of steps it uses when it runs

    M_1 = "On input string w:

    1. Scan across the tape and reject if a 0 is found to the right of a 1.
    2. Repeat if both 0s and 1s remain on the tape:
    3. Scan across the tape, crossing off a single 0 and a single 1.
    4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject. Otherwise, if neither 0s nor 1s remain on the tape, accept."

Measuring Complexity

- to analyze this TM, we need to introduce some terminology
- the number of steps an algorithm uses on a particular input may depend on several parameters
  - e.g., if the input is a graph, the number of steps may depend on the number of nodes, the number of edges, and the maximum degree of the graph, or some combination of these or other factors
  - for simplicity, we compute the run time of an algorithm as a function of the length of the input string alone
- worst-case analysis: longest running time of all inputs of a particular length
- average-case analysis: average of all running times of inputs of a particular length

Measuring Complexity

DEFINITION 7.1

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Measuring Complexity

- the exact running time of an algorithm is often a complex expression
  - we estimate it instead
- asymptotic analysis
  - run time for large inputs
  - only consider the highest order term of the expression representing the run time of the algorithm
  - disregard coefficient of that term
  - disregard all other terms
  - highest order term dominates all other terms on large input

Measuring Complexity

- asymptotic analysis
  - example: $f(n) = 6n^3 + 2n^2 + 20n + 45$
    - 4 terms
    - highest term: $6n^3$
    - disregard coefficient 6
    - f is asymptotically at most n^3
    - big-O notation: $f(n) = O(n^3)$

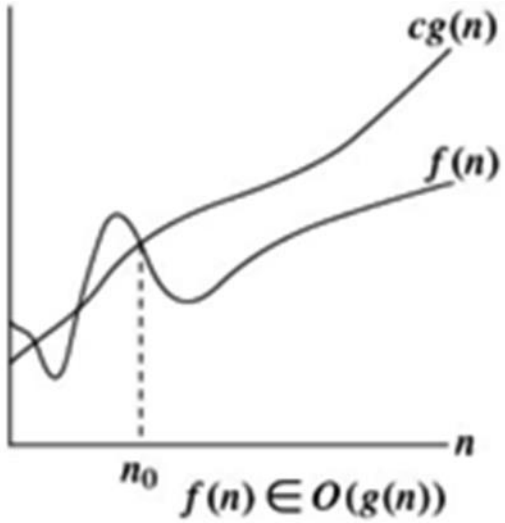
Measuring Complexity

DEFINITION 7.2

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.



Measuring Complexity

- asymptotic analysis
  - $f(n) = O(g(n))$
    - intuitively, $f \leq g$
    - disregards behavior for small n
  - example: $f(n) = 5n^3 + 2n^2 + 22n + 6$
    - $f(n) = O(n^3)$
    - formal definition:
      - let $c = 6$, $n_0 = 10$
      - $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for every $n \geq 10$
    - also, $f(n) = O(n^4)$, but $f(n) \neq O(n^2)$

Measuring Complexity

- asymptotic analysis
  - big-O works with log's in a different way
    - usually we specify the base of a log
      - e.g., $x = \log_2 n$
    - changing the base changes the value of $\log_b n$ by a constant factor
      - $\log_b n = \log_2 n / \log_2 b$
    - for $f(n) = O(\log n)$
      - do not need to specify the base since it's a constant factor
  - example: $f(n) = 3n \log_2 n + 5n \log_2 \log_2 n + 2$
    - $f(n) = O(n \log n)$

Measuring Complexity

- big-O notation can appear in expressions
  - $f(n) = O(n^2) + O(n)$
    - $O(n^2)$ dominates, so $f(n) = O(n^2)$
  - if big-O in exponent
    - $f(n) = 2^{O(n)}$
      - upper bound of 2^{cn} for some c
    - $f(n) = 2^{O(\lg n)}$
      - if $n = 2^{O(\log n)}$, $n^c = 2^{c \lg n}$
      - $n^c = O(2^{O(\log n)})$
    - $f(n) = O(n^3)$

Measuring Complexity

- polynomial bounds: n^c
- exponential bounds: 2^{n^δ} for $\delta > 0$

- small-o notation
  - big-O: $\leq$
  - small-o: $<$

Measuring Complexity

DEFINITION 7.5

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < c\,g(n)$ for all $n \geq n_0$.

Measuring Complexity

- examples:
  1. sqrt(n) = o(n)
  2. n = o(n log log n)
  3. n log log n = o(n log n)
  4. n log n = $o(n^2)$
  5. $n^2 = o(n^3)$

- f(n) = O(f(n))
- f(n) ≠ o(f(n))

Measuring Complexity

- example: $A = \{0^k1^k \mid k \geq 0\}$
  - A is context-free, so it is decidable
  - how much time does a single-tape TM need to decide A?
  - low-level TM description with head motion so that we can count the number of steps it uses when it runs

    M_1 = "On input string w:

    1. Scan across the tape and reject if a 0 is found to the right of a 1.
    2. Repeat if both 0s and 1s remain on the tape:
    3. Scan across the tape, crossing off a single 0 and a single 1.
    4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject. Otherwise, if neither 0s nor 1s remain on the tape, accept."

Measuring Complexity

- example: $A = \{0^k1^k \mid k \geq 0\}$
  - consider each stage separately
  - stage 1:
    1. Scan across the tape and reject if a 0 is found to the right of a 1.
    - verifies input 0*1*
      - n steps, where n is the length of the input
    - repositioning head at left-hand end of tape
      - n steps
    - total: 2n steps
      - O(n)
      - repositioning not mentioned in description, but OK since it only adds a constant factor

Measuring Complexity

- example: $A = \{0^k1^k \mid k \geq 0\}$
  - stages 2 and 3:
    2. Repeat if both 0s and 1s remain on the tape:
    3. Scan across the tape, crossing off a single 0 and a single 1.
  - repeatedly scans tape and crosses of a 0 and 1 on each scan
    - each scan uses $O(n)$ steps
    - cross off 2 symbols, so at most $n/2$ scans
  - total time for both stages: $(n/2)O(n) = O(n^2)$ steps

Measuring Complexity

- example: $A = \{0^k1^k \mid k \geq 0\}$
  - stage 4:
    4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject. Otherwise, if neither 0s nor 1s remain on the tape, accept.
  - single scan to decide whether to accept or reject
    - $O(n)$

Measuring Complexity

- example: $A = \{0^k1^k \mid k \geq 0\}$
  - total time for all stages
    - $O(n) + O(n^2) + O(n) = O(n^2)$
  - so, running time = $O(n^2)$

Measuring Complexity

- notation for classifying languages according to timing requirements

DEFINITION 7.7

Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, **TIME($t(n)$)**, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

- for $A = \{0^k1^k \mid k \geq 0\}$
  - $A \in$ TIME(n^2)
    - M_1 decides A in time $O(n^2)$
    - TIME(n^2) contains all languages that can be decided in $O(n^2)$ time

Measuring Complexity

- is there a machine that decides A more quickly?
  - is A in TIME($t(n)$) for $t(n) = o(n^2)$?
- try crossing off 2 0s and 1s on each scan, instead of just 1
  - reduces running time by factor of 2, so does not affect the asymptotic time
- can try a different algorithm

Measuring Complexity

- example: $A = \{0^k1^k \mid k \geq 0\}$

  M_2 = "On input string w:

  1. Scan across the tape and reject if a 0 is found to the right of a 1.
  2. Repeat if both 0s and 1s remain on the tape:
  3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, reject.
  4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
  5. If no 0s and no 1s remain on the tape, accept. Otherwise, reject."

Measuring Complexity

- does M_2 actually decide A?
  - every scan in stage 4 reduces the number of remaining 0s by half, with any remainder discarded
    - if we start with 13 0s, we end up with 6 0s
    - then 3, then 1, then 0
    - same with the number of 1s
  - even/odd parity of 0s and 1s in stage 3
    - starting with 13 0s, odd
    - 6, even
    - 3, odd
    - 1, odd
    - not performed on 0 due to stage 2 condition

Measuring Complexity

- does M_2 actually decide A?
  - even/odd parity of 0s and 1s in stage 3
    - 13: odd even odd odd
    - replace even with 0 and odd with 1, then reverse
    - corresponds to 1101 (13 in binary)
  - when stage 3 checks total number of 0s and 1s remaining is even
    - actually checking on agreement of parity of 0s and 1s
    - if all parities agree, binary representations agree
    - so, numbers are equal

Measuring Complexity

- running time of M_2
  - every stage takes $O(n)$ time
  - stages 1 and 5 are executed once
    - total: $O(n)$ time
  - stage 4 crosses off at least half 0s and 1s each iteration
    - at most: $1 + \lg n$ iterations
  - total time for stages 2, 3, and 4
    - $(1 + \lg n)\, O(n) = O(n \log n)$
  - total running time of M_2
    - $O(n) + O(n \log n) = O(n \log n)$

Measuring Complexity

- previously $A \in \text{TIME}(n^2)$
- but now $A \in \text{TIME}(n \log n)$, which is better
- cannot be further improved on TM with single tape
- any language that can be decided in $o(n \log n)$ time on single-tape TM is regular
- if the TM has a second tape, it can be decided in linear time: $O(n)$
  - copies 0s to second tape and matches against the 1s

Measuring Complexity

- example: $A = \{0^k1^k \mid k \geq 0\}$

  M_3 = "On input string w:

  1. Scan across the tape and reject if a 0 is found to the right of a 1.
  2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
  3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, reject.
  4. If all the 0s have now been crossed off, accept. If any 0s remain, reject."

Measuring Complexity

- running time of M_3
  - every stage takes $O(n)$ steps
    - total: $O(n)$ time
    - $O(n)$ is best running time possible because it takes n steps just to read the input
  - summary
    - fastest single-tape TM for A: $O(n \log n)$
    - fastest two-tape TM for A: $O(n)$

Measuring Complexity

- important difference between complexity theory and computability theory
  - computability theory
    - Church-Turing thesis implies all reasonable models of computation are equivalent
  - complexity theory
    - choice of model affects the time complexity of languages
    - e.g., languages that are decidable in linear time on one model are not necessarily decidable in linear time on another

Measuring Complexity

- with complexity theory, we classify problems according to their time complexity
  - but with which model?
  - fortunately, time requirements don't differ greatly for typical deterministic models
  - so, the choice of model isn't crucial

Measuring Complexity

- complexity relationships among models
  - how does the choice of model affect time complexity of languages?
  - three models
    - single-tape TM
    - multitape TM
    - nondeterministic TM

Measuring Complexity

- Theorem 7.8: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.
  - proof idea
    - can convert any multitape TM into a single-tape TM that simulates it
    - analyze simulation to determine how much additional time it requires
    - simulating each step on the multitape TM takes $O(t(n))$ steps on the single-tape TM
    - total time is therefore $O(t^2(n))$

Measuring Complexity

- Theorem 7.8: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.
  - proof
    - let M be a k-tape TM that runs in $t(n)$ time
    - construct TM S that runs in $O(t^2(n))$ time
    - S simulates M, as described in Theorem 3.13
      - S uses its single tape to represent all k tapes of M
      - tapes are stored consecutively, with M's heads marked at certain locations

Measuring Complexity

- Theorem 7.8: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.
  - proof (cont.)
    - initially, S puts its tape into the format that represents all tapes of k then simulates M's steps
    - to simulate one step, S scans all information stored on its tape to determine the symbols under M's tape heads
    - S makes another pass over its tape to update the tape contents and head positions
    - if one of M's heads moves rightward onto the previously unread portion of its tape, S must increase the amount of space allocated to this tape
      - shifts a portion of its own tape one cell to the right

Measuring Complexity

- Theorem 7.8: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.
  - proof (cont.)
    - analyze simulation
    - for each step of M, S makes 2 passes over the active portion of its tape
      - obtain the information to determine the next move
      - carry it out
    - length of active portion of S's tape determines how long S takes to scan it
      - to get upper bound, sum lengths of active portions of M's k tapes

Measuring Complexity

- Theorem 7.8: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.
  - proof (cont.)
    - each active portion is at most $t(n)$
      - M uses $t(n)$ tape cells in $t(n)$ steps if the head moves rightward at every step
      - fewer if head ever moves leftward
    - to simulate each of M's steps
      - S performs 2 scans and up to k rightward shifts
        - each uses $O(t(n))$ time
      - total time for S to simulate one of M's steps: $O(t(n))$

Measuring Complexity

- Theorem 7.8: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.
  - proof (cont.)
    - total time
      - initial stage (formatting): $O(n)$
      - to simulate $t(n)$ steps of M, using $O(t(n))$ steps
        - $t(n) \times O(t(n)) = O(t^2(n))$
        - each uses $O(t(n))$ time
      - total time: $O(n) + O(t^2(n))$ steps
    - running time: $O(t^2(n))$

Measuring Complexity

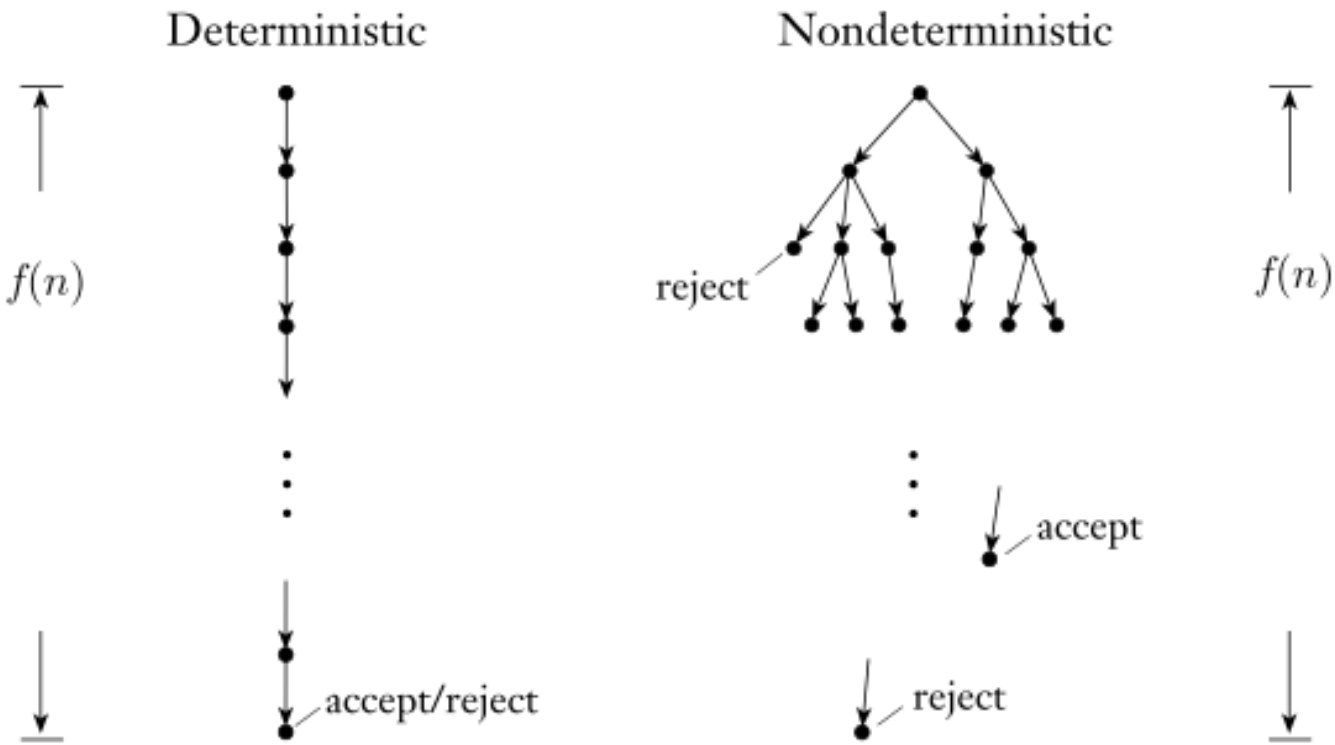
- any language that is decidable on a nondeterministic TM is decidable on a deterministic single-tape TM that requires significantly more time
  - first, we must define the running time of a nondeterministic TM, where all its computation branches halt on all inputs

DEFINITION 7.9

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

Measuring Complexity

- measuring deterministic and nondeterministic time



Measuring Complexity

- the definition of the running time of a NTM is not intended to correspond to any real-world computing device
- it is a useful mathematical definition that assists in characterizing the complexity of an important class of computational problems

Measuring Complexity

- Theorem 7.11: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.
  - proof
    - let N be a NTM running in $t(n)$ time
    - construct TM D to simulate N as in Theorem 3.16 by searching N's nondeterministic computation tree
    - analyze that simulation
      - on input of length n, every branch of N's nondeterministic computation tree has a length of at most $t(n)$
      - every node in the tree can have at most b children
        - b is the maximum number of legal choices given by the transition function
      - total number of leaves is at most $b^{t(n)}$

Measuring Complexity

- Theorem 7.11: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.
  - proof (cont.)
    - simulation proceeds through breadth first exploration
      - visits all nodes at depth d before going on to depth d+1
      - total number of nodes in tree < 2 x max number of leaves
        - $O(b^{t(n)})$
      - travel from root to node is $O(t(n))$
    - total running time of D: $O(t(n))b^{t(n)} = 2^{O(t(n))}$

Measuring Complexity

- Theorem 7.11: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.
  - proof (cont.)
    - total running time of D: $O(t(n))b^{t(n)} = 2^{O(t(n))}$
    - in Theorem 3.16, the TM D has three tapes
    - converting to a single-tape TM at most squares the running time
    - thus, the running time of the single-tape simulator is
      - $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$

The Class P

- from the previous theorems, there is at most a square or polynomial difference between deterministic single-tape and multitape TMs
- exponential difference between complexity of problems on deterministic vs. nondeterministic TMs

The Class P

- polynomial time differences considered small but exponential differences large
  - dramatic time differences between growth rates
    - e.g., for n = 1000
      - n^3 = 1B
      - 2^n > number of atoms in the universe
  - polynomial algorithms typically good enough to solve problems
  - exponential algorithms not useful

The Class P

- exponential run time often results from brute-force algorithms
  - exhaustively searching through a solution space
  - e.g., factoring a number by searching through all potential divisors
    - search space is exponential
- a deeper understanding of the problem may lead to a polynomial time algorithm

The Class P

- all reasonable deterministic models are polynomially equivalent
  - any one can simulate another with only a polynomial increase in running time
  - reasonable: in terms of run time on a computer
- our focus will be on aspects of time complexity that are unaffected by polynomial differences in run time
  - the issue is computation, not properties of particular models such as TMs
  - n vs. n^3 is a big difference practically, but we ignore such differences when considering polynomial time vs. nonpolynomial time

The Class P

DEFINITION 7.12

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

- the class P plays a central role in our theory and is important because
  - P is invariant for all models of computation that are polynomially equivalent to deterministic single-tape TMs
  - P roughly corresponds to problems that are reasonably solvable on a computer
    - runs in some n^k time
    - n^{1000} may not be practical, but any solution that goes from exponential time to polynomial time is notable

The Class P

- Example Problems in P
  - high-level description of algorithm free from any particular computational model
  - we will continue to describe algorithms with numbered stages
    - number of stages and number of steps important

The Class P

- to show an algorithm runs in polynomial time, we need to do two things
  - give polynomial upper bound using big-O notation on the number of stages that the algorithm uses on input of size n
  - examine the individual stages to ensure each can be implemented in polynomial time on a reasonable deterministic model
    - choose stages when describing algorithm
    - polynomial number of stages, each of which can be performed in polynomial time results in a polynomial

The Class P

- must also consider encoding of the problem
  - use <> to represent some reasonable encoding
  - reasonable = polynomial time encoding/decoding
    - e.g., encoding 17 as 1111111111111111 is not reasonable since it is exponentially larger than other encodings, such as base k notation for $k \geq 2$

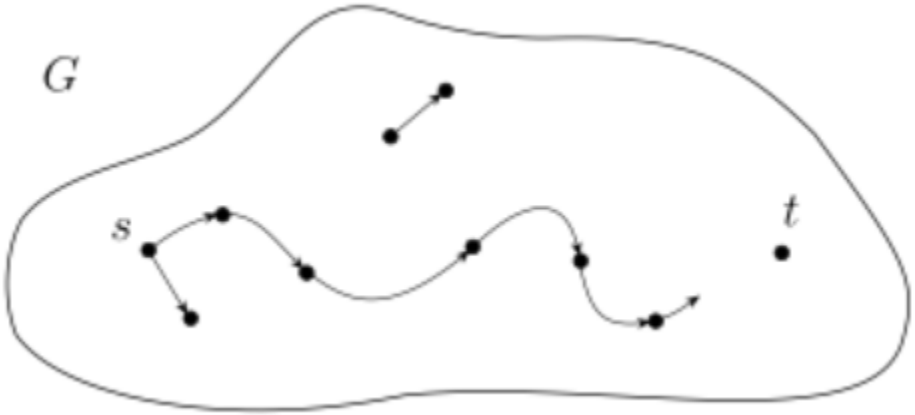
The Class P

- encoding of graphs
  - list of nodes and its edges
  - adjacency matrix
  - reasonable = run time on number of nodes vs. the size of the graph representation
    - should be polynomial on the number of nodes

The Class P

- PATH problem
  - determine whether a directed graph G contains a directed path from s to t

    PATH = {<G,s,t> | G is a directed graph that has a directed path from s to t}



The Class P

- Theorem 7.14: PATH $\in$ P
  - proof idea
    - present a polynomial time algorithm that decides PATH
    - brute-force algorithm won't work
      - e.g., examine all paths to find one from s to t
        - if m is the number of nodes in G, the path cannot be longer than m
        - all such paths = m^m, which is exponential
    - instead, use polynomial algorithm
      - breadth-first search
      - successively mark all nodes in G reachable from s by directed paths of 1, 2, 3, …, m

The Class P

- Theorem 7.14: PATH $\in$ P
  - proof
    - a polynomial time algorithm M for PATH

    M = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

    1. Place a mark on node s.
    2. Repeat the following until no more nodes are marked:
    3. Scan all the edges of G. If an edge (a,b) is found going from a marked node a to an unmarked node b, mark node b.
    4. If t is marked, accept. Otherwise, reject."

    - analyze algorithm to ensure it runs in polynomial time
      - stages 1 and 4: performed once in polynomial time
      - stage 3 runs at most m times
      - total time = 1 + 1 + m, so M is a polynomial algorithm for PATH

The Class P

- the next problem concerns relatively prime numbers
  - two numbers are relatively prime if their largest common factor is 1
    - e.g., 10 and 21 are relatively prime
    - e.g., 10 and 22 are not

RELPRIME = {<x, y> | x and y are relatively prime}

The Class P

- Theorem 7.15: RELPRIME $\in$ P
  - proof idea
    - one algorithm searches through all possible divisors of both numbers and accepts if none are greater than 1
      - magnitude of a number in binary is exponential in the length of its base
      - therefore, the brute-force algorithm searches an exponential number of divisors; hence, it is exponential
    - Euclidean algorithm for greatest common divisor
      - gcd(x,y): largest integer that evenly divides both x and y
        - e.g., gcd(18,24) = 6
        - x and y are relatively prime if gcd(x,y) = 1
      - denoted by E in algorithm
      - uses mod function

The Class P

- Theorem 7.15: RELPRIME $\in$ P
  - proof
    - Euclidean algorithm E

      E = "On input <x, y>, where x and y are natural numbers in binary:

      1. Repeat until y = 0.
      2. Assign x = x mod y.
      3. Exchange x and y.
      4. Output x."

The Class P

- Theorem 7.15: RELPRIME $\in$ P
  - proof
    - Algorithm R uses E

      R = "On input <x, y>, where x and y are natural numbers in binary:

      1. Run E on <x, y>.
      2. If the result is 1, accept. Otherwise, reject."

    - if E runs in polynomial time, so does R
    - E is known to be correct
    - need to show E runs in polynomial time

The Class P

- E runs in polynomial time
  - every execution of stage 2 reduces x by at least half
  - after stage 2, $x < y$ due to mod
  - after stage 3, $x > y$ by exchange
  - then in stage 2, if $x/2 \geq y$, then $x \bmod y < y \leq x/2$ and x is reduced by at least half
    - if $x/2 < y$, $x \bmod y = x - y < x/2$ and x is cut by at least half
  - since x and y are exchanged in stage 3, both x and y are reduced by at least half every other iteration
    - max times stages 2 and 3 executed is lesser of $2 \log_2 x$ and $2 \log_2 y$
    - both logs are proportional to lengths of representations
    - number of stages executed = O(n)
    - each stage of E uses polynomial time, so total running time is polynomial

The Class P

- Theorem 7.16: Every context-free language $\in$ P
  - proof idea
    - we know every CFL is decidable
      - Theorem 4.9 provided an algorithm that decides it
      - if algorithm runs in polynomial time, done
    - let L be a CFL generated by CFG G in Chomsky NF
      - any derivation of string w has $2n - 1$ steps, $|w| = n$
      - if any of these derives w, accepts; if not, rejects
    - this algorithm does not run in polynomial time
      - number of derivations with k steps may be exponential in k

The Class P

- Theorem 7.16: Every context-free language $\in$ P
  - proof idea
    - use dynamic programming to get a polynomial algorithm
      - uses information about subproblems to solve larger problems
      - record solutions in table
    - subproblem: determine if each variable in G generates each substring of w
      - uses n x n table to store solutions to subproblems
      - (i,j)th entry contains variables that generate $w_i w_{i+1} \ldots w_j$
      - algorithm fills table for each substring of w of length 1, 2, …
      - uses entries of shorter length for entries for longer lengths
      - e.g., longer length string is split into shorter strings
        - for each split, rule $A \rightarrow BC$ checked to see if B generates the first part of the string and C the second part, using table entries
        - if so, A generates substring and added to table
        - algorithm starts with strings of length 1, i.e., using rules of the form $A \rightarrow b$

The Class P

- Theorem 7.16: Every context-free language $\in$ P
  - let G be a CFG in Chomsky normal form for CFL L
  - proof

D = "On input $w = w_1 \ldots w_n$:

1. For $w = \varepsilon$, if $S \rightarrow \varepsilon$ is a rule, accept; else, reject.
2. For i = 1 to n: [examine each substring of length 1]
3. For each variable A:
4. Test whether $A \rightarrow b$ is a rule, where $b = w_i$.
5. If so, place A in table(i,i).
6. For l = 2 to n: [l is the length of the substring]
7. For i = 1 to n – l + 1 [i start position of substring]
8. Let j = i + l – 1 [j end position of substring]
9. For k = i to j – 1 [k split position]
10. For each rule $A \rightarrow BC$:
11. If table(i,k) contains B and table(k + 1, j) contains C, put A in table(i,j)
12. If S is in table(1,n), accept; else, reject."

The Class P

- each stage easily implemented to run in polynomial time
  - stages 4 and 5 run at most nv times, where v is the number of variables in G (constant), and run in $O(n)$
  - stage 6 runs at most n times
  - each time stage 6 runs, stage 7 runs at most n times
  - each time stage 7 runs, stages 8 and 9 run at most n times
  - each time stage 9 runs, stage 10 runs r times, where r is the number of rules of G (constant)
  - thus, stage 11, the inner loop, runs $O(n^3)$ times
  - therefore, D executes in $O(n^3)$ stages

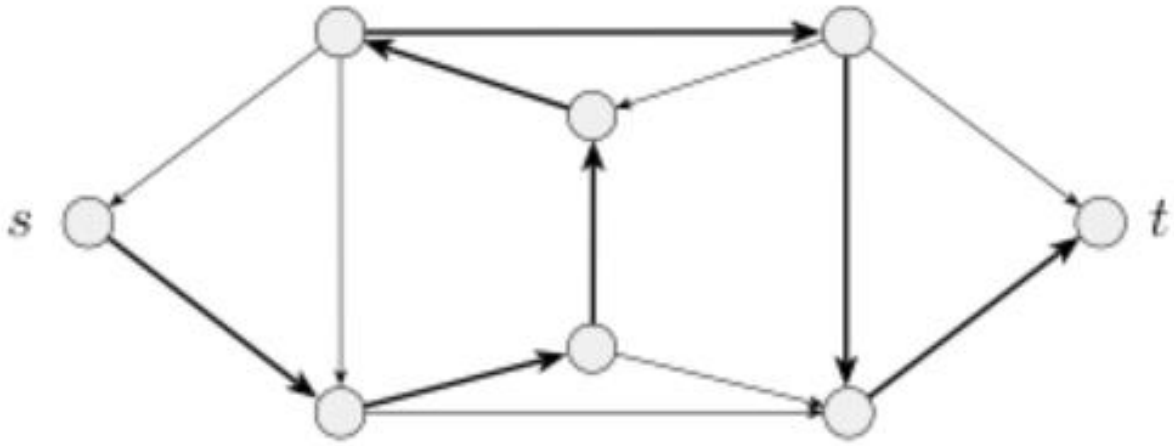
The Class NP

- we can avoid brute-force search in many problems to obtain polynomial time solutions
- avoiding brute-force approaches in other problems haven't been successful
  - so far, polynomial time algorithms for these problems are not known to exist
  - polynomial solutions may exist, but for some problems, no polynomial solution may be possible
- the complexities of many problems are linked
  - a polynomial solution for one problem may be used to solve an entire class of problems

The Class NP

- example: Hamiltonian path: directed path through a directed graph that visits each node exactly once

HAMPATH = {<G, s, t> | G is a directed graph with a Hamiltonian path from s to t}



The Class NP

- example: Hamiltonian path: directed path through a directed graph that visits each node exactly once

  HAMPATH = {<G, s, t> | G is a directed graph with a Hamiltonian path from s to t}

  - exponential algorithm for HAMPATH using brute-force algorithm for PATH (Theorem 7.14)
    - just add a check to verify the path is Hamiltonian

  - no one knows if HAMPATH is solvable in polynomial time

The Class NP

- polynomial verifiability
  - ability to verify if a solution is correct in polynomial time
  - even if we cannot actually determine if a solution exists in polynomial time
  - HAMPATH is polynomially verifiable

The Class NP

- example: show that a natural number is a composite (non-prime)

  COMPOSITES = {x | x = pq for integers p,q > 1}

  - easy to verify a number is composite
    - just find a divisor
    - but this is non-polynomial
  - COMPOSITES is polynomially verifiable since relatively recent, but more complicated, polynomial time algorithm was discovered

The Class NP

- some problems may not be polynomially verifiable
  - example: $\overline{\text{HAMPATH}}$, the complement of HAMPATH
    - even if $\overline{\text{HAMPATH}}$ were solved, no good way to verify its nonexistence without using the same exponential algorithm used to solve it

The Class NP

DEFINITION 7.18

A ***verifier*** for a language A is an algorithm V, where

$$A = \{w | \ V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

- verifier uses additional information: c in the definition to verify that a string is a member of A
  - called the certificate, or proof, of membership
  - for polynomial verifiers, certificate has polynomial length (in w)

The Class NP

- for HAMPATH, a certificate would be a Hamiltonian path from s to t
- for COMPOSITES, a certificate would be one of its divisors
- the verifier can check in polynomial time that the input is in the language when given the certificate

The Class NP

DEFINITION 7.19

NP is the class of languages that have polynomial time verifiers.

- class NP is important because it contains many problems of practical interest
  - HAMPATH
  - COMPOSITES
    - also a member of P, a subset of NP, but proving is harder
- NP: nondeterministic polynomial time
  - using nondeterministic polynomial time Turing machines

The Class NP

- NTM to decide HAMPATH in nondeterministic polynomial time
  - time of a nondeterministic machine is the time used by the longest computational branch (Definition 7.9)

N_1 = "on input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

1. Write a list of numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, reject. Otherwise, all tests have been passed, so accept."

  - each stage runs in polynomial time, so this algorithm runs in nondeterministic polynomial time

The Class NP

- Theorem 7.20: *A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.*
  - proof idea
    - show how to convert a polynomial time verifier to an equivalent NTM and vice versa
      - NTM simulates verifier by guessing the certificate
      - verifier simulates NTM by using the accepting branch as the certificate

The Class NP

- Theorem 7.20: *A* language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.
  - proof
    - let $A \in$ NP and show A is decided by a polynomial time NTM N
    - let V be the polynomial time verifier for A
      - TM that runs in time n^k

N = "on input w of length n:

1. Nondeterministically select string c of length at most n^k.
2. Run V on input <w,c>.
3. If V accepts, accept; otherwise, reject."

The Class NP

- Theorem 7.20: *A* language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.
  - proof
    - other direction
    - assume *A* is decided by polynomial time NTM N
    - construct a polynomial time verifier V

V = "on input <w,c> where w and c are strings:

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step (as in Theorem 3.16).
2. If this branch of N's computation accepts, accept; otherwise, reject."

The Class NP

DEFINITION 7.21

$\mathbf{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$

- analogous to deterministic time complexity class TIME(t(n))

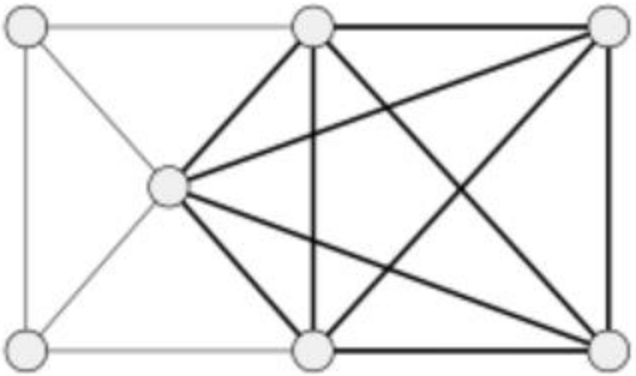
COROLLARY 7.22

$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k).$

- NP insensitive to choice of reasonable nondeterministic computational model
- each stage in nondeterministic polynomial time algorithm must have an obvious polynomial time
  - every branch uses at most polynomial many stages

The Class NP

- example NP problems
  - clique problem
    - clique: subgraph in an undirected graph where every two nodes are connected by an edge
    - k-clique: clique that contains k nodes
      - 5-clique



The Class NP

- example NP problems
  - clique problem: determine if a graph contains a clique of a specified size

    CLIQUE = {<G,k> | G is undirected graph with k-clique}

  - Theorem 7.24: CLIQUE is in NP
    - proof idea: the clique is the certificate

The Class NP

- Theorem 7.24: CLIQUE is in NP
  - proof
    - verifier V for CLIQUE

      V = "on input <<G,k>,c>:

      1. Test whether c is a subgraph with k nodes in G.
      2. Test whether G contains all edges connecting nodes in c.
      3. If both pass, accept; otherwise, reject."

The Class NP

- Theorem 7.24: CLIQUE is in NP
  - alternative proof using nondeterministic polynomial time Turing machine
    - NTM N for CLIQUE

      N = "on input <G,k>, where G is a graph:

      1. Nondeterministically select a subset c of k nodes in G.
      2. Test whether G contains all edges connecting nodes in c.
      3. If yes, accept; otherwise, reject."

The Class NP

- example NP problems
  - subset sum: given a collection of numbers $x_1, \ldots, x_k$ and a target number t, determine whether the collection contains a subcollection that adds up to t

    SUBSET-SUM = $\{\langle S,t\rangle \mid S = \{x_1, \ldots, x_k\}$ and for some $\{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \ \Sigma y_i = t\}$

  - example: $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ SUBSET-SUM because 4 + 21 = 25
    - note that $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ are multisets, so elements can be repeated

The Class NP

- Theorem 7.25: SUBSET-SUM is in NP
  - proof
    - the subset is the certificate
    - verifier V for SUBSET-SUM

      V = "on input <<S,t>,c>:

      1. Test whether c is a collection of numbers that sum to t.
      2. Test whether S contains all the numbers in c.
      3. If both pass, accept; otherwise, reject."

The Class NP

- Theorem 7.25: SUBSET-SUM is in NP
  - alternative proof using nondeterministic polynomial time Turing machine

    N = "on input <S,t>:

    1. Nondeterministically select a subset c of the numbers in S.
    2. Test whether c is a collection of numbers that sum to t.
    3. If the test passes, accept; otherwise, reject."

The Class NP

- complements of these sets, $\overline{\text{CLIQUE}}$ and $\overline{\text{SUBSET-SUM}}$ are not obviously members of NP
  - verifying that something is not present seems to be more difficult than verifying that it is preset
  - separate class, coNP
    - contains languages that are complements of languages in NP
    - unknown whether coNP is different from NP

The Class NP

- P vs. NP
  - P = class of languages for which membership can be decided quickly
  - NP = class of languages for which membership can be verified quickly
    - quickly = polynomial time
  - HAMPATH and CLIQUE are members of NP, but are not known to be in P
  - P and NP could be equal!

The Class NP

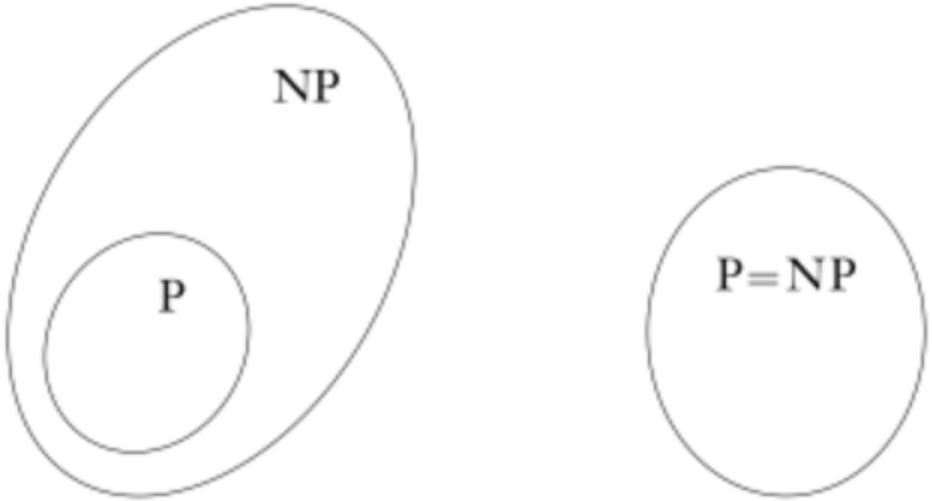
- P = NP is one of the greatest unsolved problems in computer science
  - if P = NP, any polynomially verifiable problem would be polynomially decidable
  - most researchers believe they are not equal because much time has been spent trying to find polynomial solutions to NP problems without success
  - proving the classes are unequal is beyond scientific reach, as it would entail showing that no fast algorithm exists to replace brute-force search
  - best deterministic method known for deciding languages in NP uses exponential time, so we can prove

$$\mathbf{NP} \subseteq \mathbf{EXPTIME} = \bigcup_k \mathbf{TIME}(2^{n^k})$$

    - but don't know if NP is contained in a smaller class

The Class NP

- one of these possibilities is correct



NP-Completeness

- NP-complete problems are important because if a polynomial time algorithm is found for one problem, all problems in NP would be solvable in polynomial time
  - this would show P = NP
  - similarly, if any problem in NP is proven to require more than polynomial time, all others would, too
  - in general, most believe that P ≠ NP, so showing a problem is NP-complete also suggests that no polynomial time algorithm will be found

NP-Completeness

- one of the simplest NP-complete problems is satisfiability
  - Boolean variables connected with AND, OR, and NOT
  - a Boolean formula is satisfiable if some assignment of 0s and 1s makes the formula evaluate to 1 (TRUE)
  - e.g., for $\varphi = (\neg x \wedge y) \vee (x \wedge \neg z)$
    - $x = 0$, $y = 1$, and $z = 0$ satisfies φ

SAT = {<φ> | φ is a satisfiable Boolean formula}

SAT ∈ P iff P = NP

NP-Completeness

- polynomial time reducibility
  - when A reduces to B, a solution to B can be used to solve A

DEFINITION 7.29

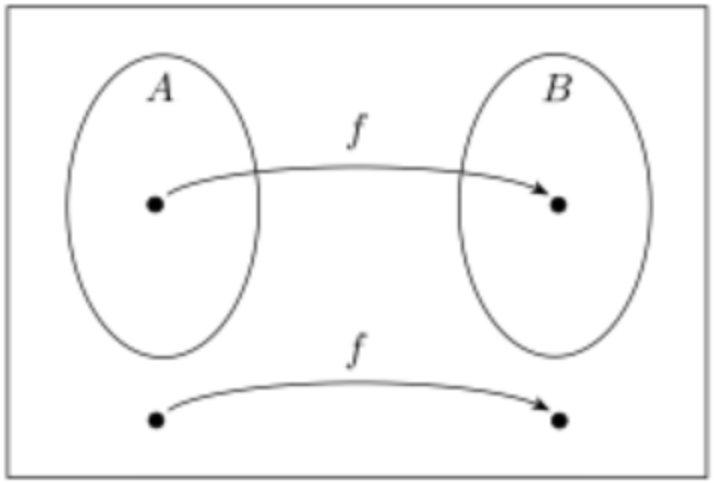
Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_P B$, if a polynomial time computable function $f: \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

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

NP-Completeness

- polynomial time reducibility
  - to test whether $w \in A$, use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$



NP-Completeness

- polynomial time reducibility
  - if one language is polynomial time reducible to a language already known to be have a polynomial time solution, we obtain a polynomial time solution to the original language

NP-Completeness

- polynomial time reducibility
  - Theorem 7.31: if $A \leq_p B$ and $B \in P$, then $A \in P$
    - proof: let M be a polynomial algorithm deciding B and let f be the polynomial reduction from A to B, then a polynomial algorithm N decides A

      N = "On input w:
      1. Compute f(w).
      2. Run M on input f(w) and output whatever M outputs. "

    - f is a reduction from A to B, so $w \in A$ when $f(w) \in B$
    - M accepts f(w) whenever $w \in A$
    - N runs in polynomial time because its stages do so

NP-Completeness

- 3SAT
  - literal: Boolean variable (x) or negated Boolean variable
  - clause: several literals connected with or's ($\vee$)
  - conjunctive normal form (cnf): several clauses connected with and's ($\wedge$)
    - example: $(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6)$
  - 3cnf-formula: all the clauses have 3 literals
    - example: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6 \vee x_4)$
  - if an assignment satisfies the formula, each clause must contain at least one literal that evaluates to 1

3SAT = {<φ> | φ is a satisfiable 3cnf-formula}

NP-Completeness

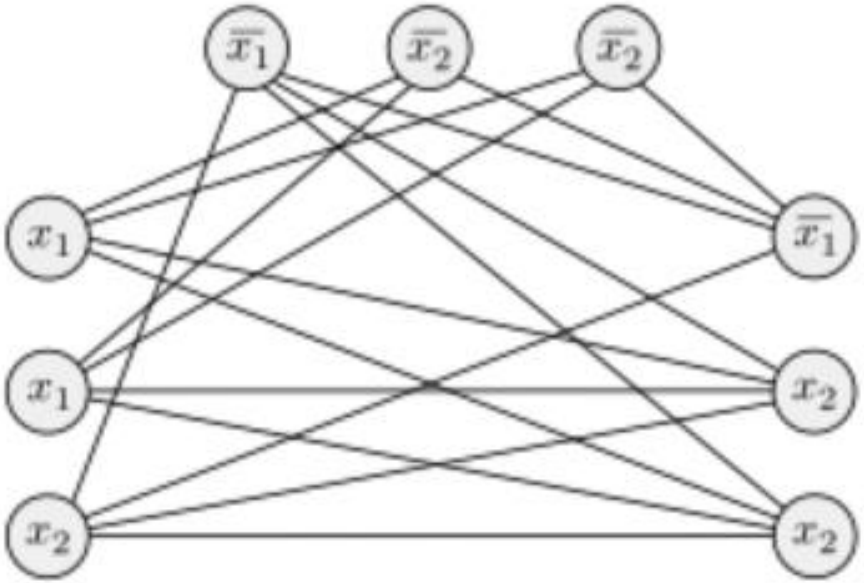
- Thm 7.32: 3SAT is polynomial time reducible to CLIQUE
  - proof idea
    - convert formulas to graphs
    - in constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula
    - structures in the graph are designed to mimic the behavior of the variables and classes

NP-Completeness

- Thm 7.32: 3SAT is polynomial time reducible to CLIQUE
  - proof
    - let $\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$
    - the reduction f generates the string <G,k>
    - nodes of G are organized into k groups of three nodes each called triples $t_1, \ldots, t_k$
    - each triple corresponds to one of the clauses in φ
    - each node in a triple corresponds to a literal in the clause
    - label each node in G with its corresponding literal in φ
    - the edges in G connect all pairs of nodes except
      - nodes in the same triple
      - two nodes with contradictory labels, e.g., x_2 and $\neg x_2$

NP-Completeness

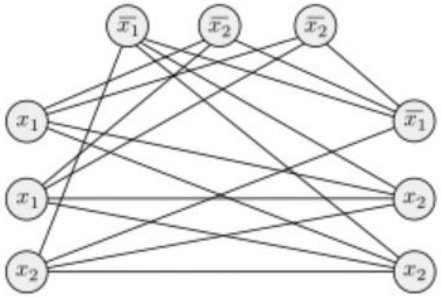
- Thm 7.32: 3SAT is polynomial time reducible to CLIQUE
  - proof
    - ex: $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$



    - now show φ is satisfiable iff G has a k-clique

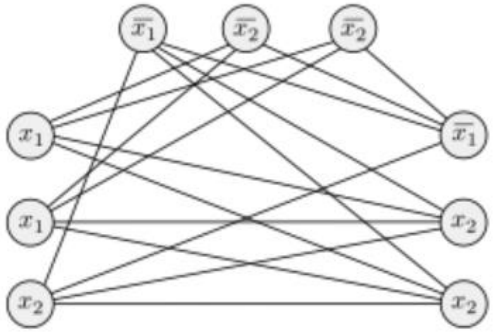
NP-Completeness

- Thm 7.32: 3SAT is polynomial time reducible to CLIQUE
  - proof
    - ex: $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$
    - suppose φ has a satisfying assignment
      - at least one TRUE in each clause
    - select one TRUE literal in each triple
      - if > 1, any OK
    - these nodes form a k-clique
      - k nodes – we choose one from each of k triples
      - edges selected cannot violate exceptions
      - cannot be from same triple since we selected only one per triple
      - no contradictory labels



NP-Completeness

- Thm 7.32: 3SAT is polynomial time reducible to CLIQUE
  - proof
    - ex: $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$
    - suppose G has a k-clique
      - no two of the clique's nodes occur in the same triple because no edges connect nodes in the same triple
      - each of the k triples contains exactly one of the k-clique nodes



    - assign truth values to variables of φ so that each literal labeling a clique node is made TRUE
      - always possible since contradictory nodes not connected by an edge, and therefore won't be in the same clique
      - this assignment of variables satisfies φ because each triple contains a clique node and hence each clause contains a literal that is TRUE; hence, φ is satisfiable

NP-Completeness

- Theorems 7.31 and 7.32 tell us that if CLIQUE is solvable in polynomial time, so is 3SAT
  - remarkable since problems seem different
    - polynomial time reducibility links their complexities
  - CLIQUE is also NP-complete

NP-Completeness

- Definition 7.34: A language B is NP-complete if it satisfies two conditions:
  1. B is in NP
  2. every A in NP is polynomial time reducible to B

- Theorem 7.35: If B is NP-complete and $B \in P$, then P = NP

NP-Completeness

- Theorem 7.36: If B is NP-complete and B $\leq_p$ C for C in NP, then C is NP-complete
  - proof
    - C is in NP (given)
    - must show every A in NP is polynomial time reducible to C
    - since B is NP-complete, every language is polynomial time reducible to B, and B is polynomial time reducible to C
    - if A is polynomial time reducible to B, and B to C, then A is polynomial time reducible to C
    - therefore, every language in NP is polynomial time reducible to C

NP-Completeness

- once we have one NP-complete problem, we may obtain others by polynomial time reduction
- establishing first NP-complete problem difficult
- start with SAT

NP-Completeness

- Theorem 7.37: SAT is NP-complete
  - proof idea
    - easy to show SAT in NP
    - harder to show any language in NP is polynomial time reducible to SAT
    - need to construct polynomial time reduction for each language A in NP to SAT
    - reduction for A takes a string w and produces a Boolean formula φ that simulates the NP machine for A on input w
      - if the machine accepts, φ has satisfying values
      - otherwise, no satisfying assignment of values
      - $w \in A$ iff φ is satisfiable

NP-Completeness

- Theorem 7.37: SAT is NP-complete
  - proof idea (cont.)
    - constructing reduction to work in this way not difficult
    - but many details
    - Boolean formulas contain AND, OR, and NOT
      - similar to circuitry in computers
    - hence, the fact that we can design a Boolean formula to simulate a Turing machine should not be surprising
    - this (long) proof is used to show the Cook-Levin Theorem

NP-Completeness

- Corollary 7.42: 3SAT is NP-complete
  - proof
    - 3SAT is in NP
    - need to prove all languages in NP reduce to 3SAT in polynomial time
      - could show SAT polynomial reduces to 3SAT
      - instead, modify long proof to produce a formula in conjunctive normal form with three literals per clause

Additional NP-Complete Problems

- NP-complete problems appear in many fields
  - most naturally occurring NP problems are known to be either in P or to be NP-complete
    - for new problem, try to show NP-complete so that time is not wasted trying to develop an algorithm that may not exist
  - to show languages are NP-complete
    - show a polynomial time reduction from 3SAT to the language
    - other NP-complete problems could be used instead
    - when constructing the reduction, look for structures that can simulate variables and clauses in Boolean formulas
      - structures called gadgets

Additional NP-Complete Problems

- vertex cover
  - G: undirected graph
  - vertex cover: a subset of nodes where every edge of G touches one of those nodes
  - vertex cover problem asks whether a graph contains a vertex cover of a specified size

VERTEX-COVER = {<G,k> | G is an undirected graph that has a k-node vertex cover}

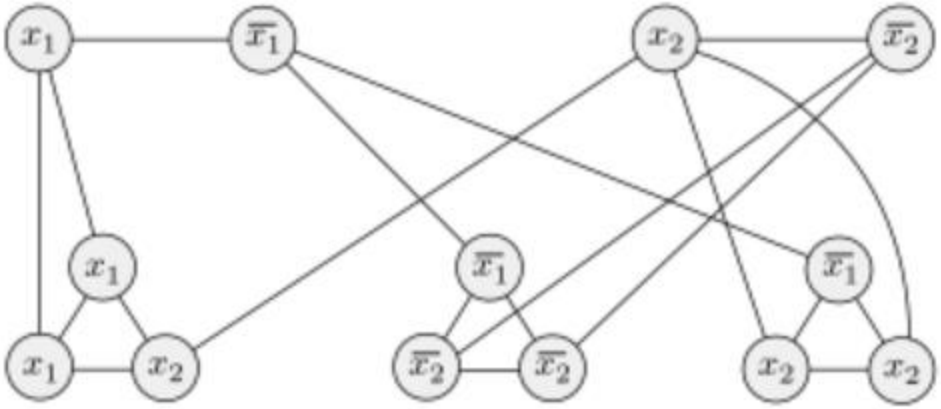
NP-Completeness

- Theorem 7.44: VERTEX-COVER is NP-complete
  - proof idea
    - show that is in NP and all NP-problems are polynomial reducible to VERTEX-COVER
      - first part: a certificate is simply a vertex cover of size k
      - second part: show 3SAT is polynomial time reducible to VERTEX-COVER
        - convert 3cnf-formula φ into a graph G and a number k, so that φ is satisfiable whenever G has a vertex cover with k nodes
        - G simulates φ

NP-Completeness

- Theorem 7.44: VERTEX-COVER is NP-complete
  - proof
    - graph from reduction produced from

$$\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$



NP-Completeness

- other NP-complete problems
  - Theorem 7.46: HAMPATH is NP-complete
  - Theorem 7.55: UHAMPATH is NP-complete
  - Theorem 7.56: SUBSET-SUM is NP-complete