Theory of Computation
(Algorithmically Hard Problems)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

August 10, 2023

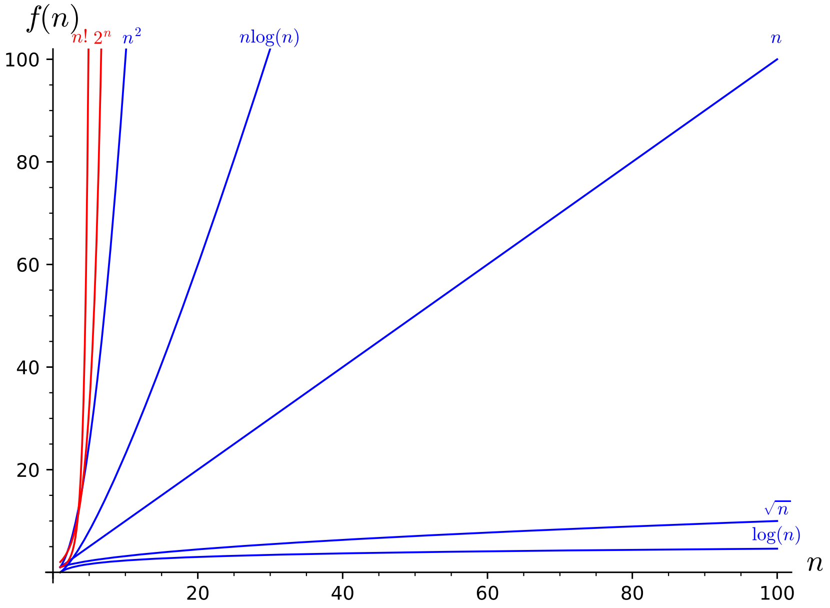
Algorithmically unsolvable problems

Problem	Running time
Simulate problem	∞
Halting problem	∞
Program correctness	∞
Program equivalence	∞
Integral roots of a polynomial	∞
Goodstein's theorem	∞
Generalized $(3n+1)$ problem	∞
Post correspondence problem	∞

Algorithmically solvable problems

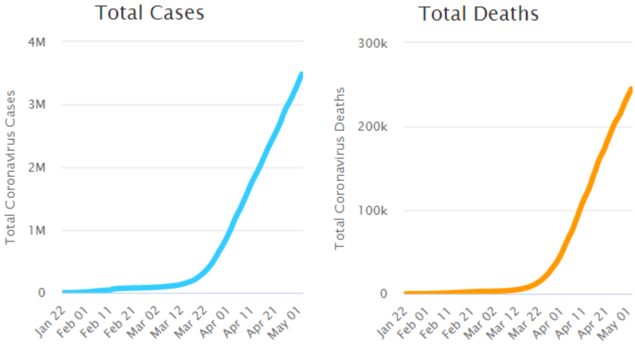
Problem	Running time
Search in a sorted array	$\mathcal{O}(\log n)$
Search in an unsorted array	$\mathcal{O}(n)$
Integer addition	$\mathcal{O}(n)$
Generate primes	$\mathcal{O}(n \log \log n)$
Sorting	$\mathcal{O}(n \log n)$
Fast Fourier transform	$\mathcal{O}(n \log n)$
Integer multiplication	$\mathcal{O}(n^2)$
Matrix multiplication	$\mathcal{O}(n^3)$
Linear programming	$\mathcal{O}(n^{3.5})$
Primality test	$\mathcal{O}(n^{10})$
Satisfiability problem	$\mathcal{O}(2^n)$
Traveling salesperson problem	$\mathcal{O}((n-1)!)$
Sudoku, chess, checkers, go	expo. class

Polynomial and exponential functions



Exponential functions

- Moore's law (Doubling of computing power every 18 months)
- Compound interest in banks
- Coronavirus



Source: `https://www.worldometers.info/coronavirus/`

Goal

Goal

- Our goal is to solve all computational problems efficiently
- An efficient/fast algorithm is one that solves a problem in polynomial time

Polynomial-time algorithm

Definition

- A polynomial-time algorithm is an algorithm whose worst-case time complexity is bounded above by a polynomial function in input size.
- If n is the input size, then there exists a polynomial $p(n)$ such that

$$T(n) \in \mathcal{O}\left(p(n)\right)$$

Analysis

- $n \log n$ is not polynomial in n but $n \log n \in \mathcal{O}\left(n^2\right)$
  Hence, algorithm with this complexity is a polynomial-time algorithm

Input and output sizes

Definition

- For a given algorithm, the input and output sizes are defined as the number of characters required to write/encode/specify the input and output, respectively, using a reasonable encoding method.
- Reasonable encodings: base 2, base 16, base 10, base $b \geq 2$
  Unreasonable encoding: base 1 (i.e., unary encoding)

Example

- Problem: $\textsc{Sort}(a[1..n])$
  Input: n positive integers
  Output: n numbers in nondecreasing order. Then
  Suppose $L = \max(a[1..n])$
  Input size: $\Theta\left(n \log L\right)$
  Output size: $\Theta\left(n \log L\right)$

Input and output sizes

IsPrime(n)

$answer \leftarrow true$
for $i \leftarrow 2$ to $\lfloor \sqrt{n} \rfloor$ do
  if n is divisible by i then
    $answer \leftarrow false$
    break
return $answer$

Problem

- Time complexity of IsPrime(n) is $\Omega\left(\sqrt{n}\right)$.
  Is IsPrime(n) a polynomial-time algorithm?

Solution

- No!
- Input size: $s = \log_2 n$ bits (to store value n)
  Output size: 1 bit (to store Boolean answer)
  Time complexity: $\Omega\left(\sqrt{n}\right) = \Omega\left(2^{s/2}\right)$ exponential
  But this does not prove that the problem cannot have any fast algorithm.

Input and output sizes

Problem

- Time complexity of $\textsc{Fibonacci-DP}(n)$ is $\Theta\left(n^2\right)$.
  Is $\textsc{Fibonacci-DP}(n)$ a polynomial-time algorithm?

Solution

- No!
- Input size: $s = \log_2 n$ bits (to store value n)
  Output size: $\Theta\left(n\right)$ bits (as F_n requires $\Theta\left(n\right)$ bits)
  Time complexity: $\Theta\left(n^2\right) = \Omega\left(4^s\right)$ exponential
  There cannot be any polynomial-time algorithm for computing the nth Fibonacci number. Why?
  Output size itself is exponential in the size of input.

Problems having exponential-sized output

Problem

- Problem: Print all simple paths
- Input: Graph G, source vertex x, destination vertex y
- Output: Print all simple paths from x to y

Analysis

- Output size: Worst-case exponential function of the input size
  Hence, polynomial-time algorithms don't exist
- We will not consider problems having exponential-sized output because no polynomial-time algorithms exist for such problems

Intractable problems

Definition

- A problem is intractable if an exponential amount of time is needed to discover its solution, given that the output size a polynomial function of the input size.

Example

- Problem: Equivalence of two regular expressions
  Input: Two regular expressions R_1 and R_2
  Output: Yes/no if R_1 is equivalent R_2

Analysis

- Output size: Polynomial function of input size
- There does not exist polynomial-time algorithms

Types of problems

Definitions

- A decision problem asks for a yes/no answer.
- A search problem asks for arbitrary string(s) as output.
- A counting problem asks for the number of solutions to a search problem.
- An optimization problem asks for for the best possible solution to a search problem.
- A function problem asks for a unique output for every input.

Examples

- Decision problem: $\textsc{IsPrime}(n)$
- Search problem: $\textsc{FindFactors}(n)$
- Counting problem: $\textsc{CountFactors}(n)$
- Optimization problem: $\mathrm{TSP}(G, w)$
- Function problem: $\mathrm{TSP}(G, w)$

Hardness of problems

Types

- Hard (or intractable): Problems that can never be solved in polynomial time.
- Easy: Problems that can be solved in polynomial time.
- Possibly hard (or possibly intractable): Problems that have no known polynomial time algorithms.

Examples

- Hard: Given two regular expressions R_1 and R_2, is R_1 equivalent to R_2?
- Easy: Is there a path from x to y with weight $\leq M$?
- Possibly hard: Is there a path from x to y with weight $\geq M$?

Complexity class P

Definition

- The complexity class P denotes the set of all decision problems that can be solved by deterministic algorithms in polynomial time.

Examples

- Is a given array sorted?
- Is a given graph cyclic?
- Is a given graph connected?
- Does a given set contain a specific element?
- Most problems we have seen have a corresponding decision version.

Complexity class NP

Definition

- The complexity class NP denotes the set of all decision problems that can be solved by nondeterministic algorithms in polynomial time.

Examples

- All problems in P, i.e., P $\subseteq$ NP
- Problem: Decision version of TSP(G, w, b)
  Input: Graph G, weight function w, length b
  Output: Yes if there exists a sequence of vertices (starting from a vertex and visiting each vertex exactly once) with length at most b.

Complexity class NP

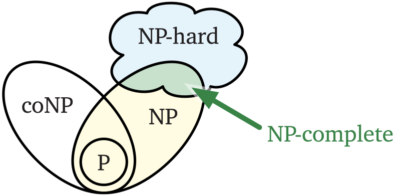
Analysis

- A nondeterministic algorithm has two stages:
  (1) (Nondeterministic) Guessing stage:
  Make all guesses simultaneously.
  (Analogous to parallel algorithm or parallel universe model)
  (2) (Deterministic) Verification stage:
  Verify/check if the guess is a correct solution or not.
- Guessing stage takes $\mathcal{O}(1)$ time
  Verification stage takes polynomial time

Complexity class NP

Definition

- A polynomial-time nondeterministic algorithm is a nondeterministic algorithm whose verification stage is a polynomial-time algorithm.
- The complexity class NP denotes the set of all decision problems that can be solved by polynomial-time nondeterministic algorithms.



Source: Jeff Erickson's Algorithms textbook

Complexity class NP

Definition

- The complexity class NP denotes the set of all decision problems with the following property: If the answer is yes, then there is a proof of this fact that can be checked in polynomial time.

  Intuitively, the complexity class NP denotes the set of all decision problems where we can verify a yes answer quickly if we have the solution in front of us.
- The complexity class co-NP denotes the set of all decision problems with the following property: If the answer is no, then there is a proof of this fact that can be checked in polynomial time.

Is P = NP?

Problem

- Is P = NP?

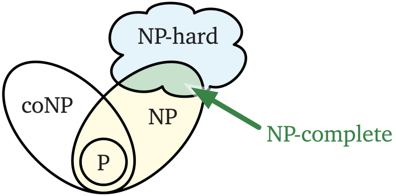
Analysis

- This is the greatest question in theoretical computer science.
- That is:
  Nobody knows if (deterministic) polynomial-time algorithms exist for solving all of NP problems.
  Nobody knows if there is an NP problem that is not in P.
  Nobody knows if NP is the same set as coNP.
- Most scientists believe that P $\neq$ NP.
- It is most likely that Turing Award (i.e., the Nobel prize of computer science) will be given to the person who resolves the P $\neq$ NP problem.

Complexity classes NP-hard and NP-Complete

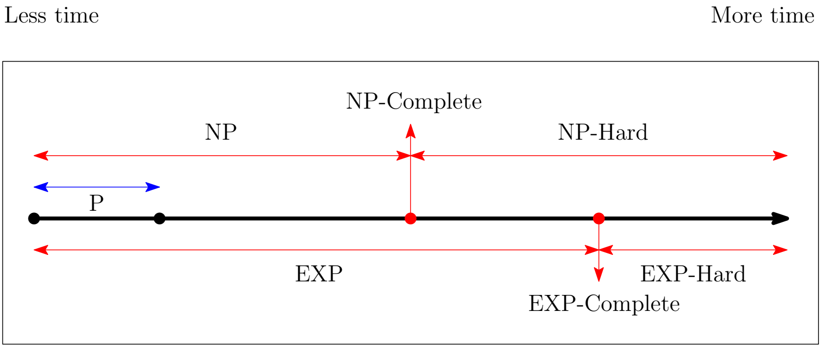
Definition

- NP = Problems solvable in poly time using nondeterminism = Problems with solutions that can be verified/checked in polynomial time.
- NP-Hard = Problems at least as hard as NP problems. Formally, a problem X is NP-Hard if every NP problem Y is polynomial-time reducible to X.
- NP-Complete = Hardest problems in NP. Formally, a problem X is NP-Complete if (i) X is in NP, and (ii) X is NP-Hard.



Source: Jeff Erickson's Algorithms textbook

Complexity classes NP-hard and NP-Complete



Easy problems and possibly hard problems

Easy problems	Possibly hard problems
Shortest path	Longest path
Linear programming	Integer linear programming
Minimum spanning tree	Traveling salesperson
2-Satisfiability	3-Satisfiability
Min cut	Max cut
Planar 4-colorability	Planar 3-colorability
Independent set on trees	Independent set

- The problems on the right have escaped efficient algorithms for decades to centuries. Why?
- The problems on the right seem hard for the same reason – they are all related.
- Each pair of those problems can be reduced to each other.

What is polynomial-time reduction?

Definition

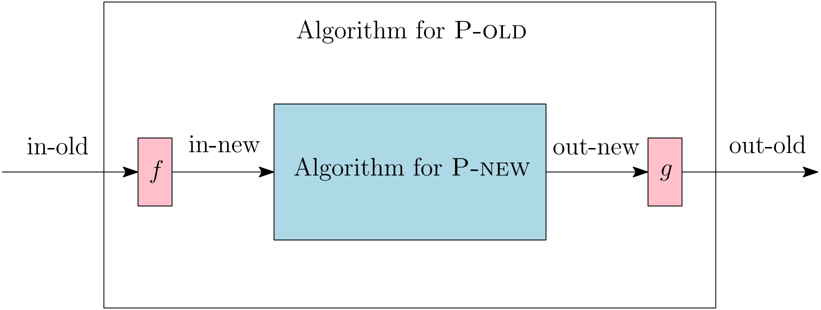
- Reduction is a fantastic idea to solve one problem using the solution to another.
- Problem P_{old} poly.-time reduces to problem P_{new}, denoted by $P_{\text{old}} \leq_p P_{\text{new}}$, if the following transformation happens in polynomial time.
  - transform any input instance of P_{old} to an instance of P_{new}
  - solve P_{new}
  - transform output of P_{new} to output of P_{old}
  - return output of P_{old}

What is polynomial-time reduction?

Definition

- Reduction is a fantastic idea to solve one problem using the solution to another.
- Problem P_{old} poly.-time reduces to problem P_{new}, denoted by $P_{\text{old}} \leq_p P_{\text{new}}$, if any instance of problem P_{old} can be solved using the following:
  - (i) poly. number of standard computational steps.
  - (ii) poly. number of calls to function that solves problem P_{new}.
- $P_{\text{old}} \leq_p P_{\text{new}}$ means P_{new} is at least as hard as P_{old}.

What is polynomial-time reduction?



PROBLEM-OLD(input-old) $\triangleright$ $P_{\text{old}} \leq_p P_{\text{new}}$

1. input-new $\leftarrow f$(input-old) $\triangleright$ poly. time transformation
2. output-new $\leftarrow$ PROBLEM-NEW(input-new)
3. output-old $\leftarrow g$(output-new) $\triangleright$ poly. time transformation
4. return output-old

PROBLEM-OLD poly. time reduces to PROBLEM-NEW

What is polynomial-time reduction?

Suppose $P_{\text{old}} \leq_p P_{\text{new}}$

Easy problems

- If P_{new} can be solved in polynomial time, then P_{old} can be solved in polynomial time.

Hard problems

- If P_{old} cannot be solved in polynomial time, then P_{new} cannot be solved in polynomial time.

Same complexity class

- If $P_{\text{new}} \leq_p P_{\text{old}}$, then P_{old} can be solved in polynomial time if and only if P_{new} can be solved in polynomial time.

Reduction: Lower and upper bounds

Suppose $P_{\text{old}} \xrightarrow{\mathcal{O}(f(n))} P_{\text{new}}$

Upper bound theorem

- If P_{new} is solvable is $\mathcal{O}\left(g(n)\right)$,
  then P_{old} is solvable in $\mathcal{O}\left(f(n) + g(n)\right)$

Lower bound theorem

- If P_{old} is solvable in $\Omega\left(g(n)\right)$ and $f(n) \in o\left(g(n)\right)$,
  then P_{new} is solvable in $\Omega\left(g(n)\right)$

Reduction: LCM $\rightarrow$ GCD

Problem

- Problem: Least common multiple (LCM)
  Input: Two integers a and b.
  Output: Return the smallest integer m such that m is a multiple of a and m is also a multiple of b.

Problem

- Problem: Greatest common divisor (GCD)
  Input: Two integers a and b.
  Output: Return the largest integer d such that d divides a and d divides b.

Reduction: LCM $\to$ GCD

$\text{LCM}(a, b)$
1. return $\frac{a \times b}{\text{GCD}(a,b)}$

GCD is poly. time $\Rightarrow$ LCM is poly. time

Reduction: DecimalCalculator $\rightarrow$ BinaryCalculator

Problem

- Problem: Arithmetic operations on decimal numbers
  Input: Two decimal numbers a and b.
  Output: Return the result of an arithmetic operation on a and b in the decimal system.

Problem

- Problem: Arithmetic operations on binary numbers
  Input: Two binary numbers a and b.
  Output: Return the result of an arithmetic operation on a and b in the binary system.

Reduction: DecimalCalculator $\rightarrow$ BinaryCalculator

$\textsc{DecimalCalculator}(a, b)$

1. $a_{\mathsf{binary}} \leftarrow \textsc{DecimalToBinary}(a)$
2. $b_{\mathsf{binary}} \leftarrow \textsc{DecimalToBinary}(b)$
3. $c_{\mathsf{binary}} \leftarrow \textsc{BinaryCalculator}(a_{\mathsf{binary}}, b_{\mathsf{binary}})$
4. $c \leftarrow \textsc{BinaryToDecimal}(c_{\mathsf{binary}})$
5. return c

$\textsc{BinaryCalculator}$ is poly. time
$\Rightarrow$ $\textsc{DecimalCalculator}$ is poly. time

Reduction: ClosestPair $\to$ Sort

Problem

- Problem: Closest pair
  Input: A set S of n numbers, and threshold t.
  Output: Is there a pair $s_i, s_j \in S$ such that $|s_i - s_j| \leq t$?

$\textsc{ClosestPair}(S, t)$

1. $\textsc{Sort}(S)$
2. return $(\min_{i \in [1,n-1]} |s_i - s_j|) \leq t$

$\textsc{Sort}$ is poly. time $\Rightarrow$ $\textsc{ClosestPair}$ is poly. time

Reduction: LIS $\rightarrow$ LCS

Problem

- Problem: Longest increasing subsequence
  Input: An integer or character sequence S.
  Output: What is the longest sequence of integer positions $\{p_1, \ldots, p_m\}$ such that $p_i < p_{i+1}$ and $S_{p_i} < S_{p_{i+1}}$?

Problem

- Problem: Longest common subsequence
  Input: Integer or character sequences S and T.
  Output: What is the longest subsequence that is common to both S to T?

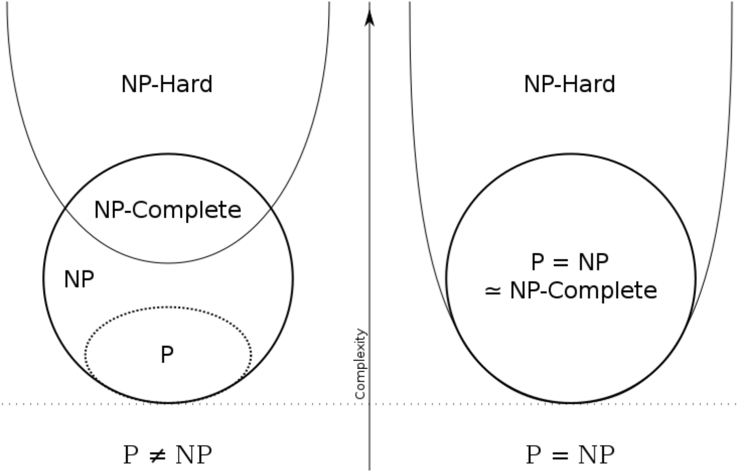
Reduction: LIS $\rightarrow$ LCS

$\mathrm{LIS}(S)$

1. $T \leftarrow \textsc{Sort}(S)$
2. $lis \leftarrow \mathrm{LCS}(S, T)$
3. return lis

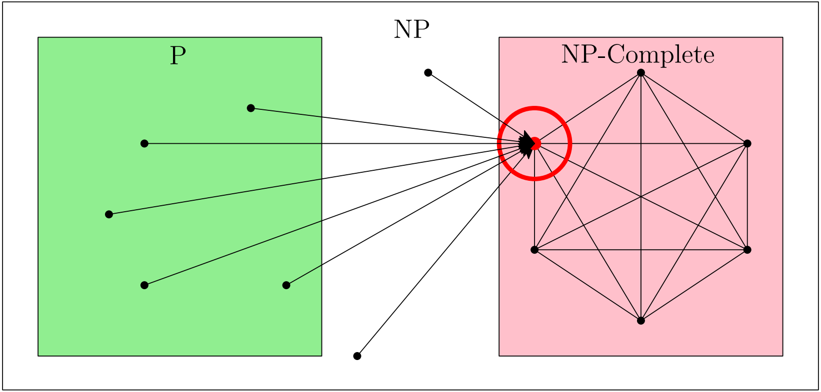
LCS is poly. time
$\Rightarrow \mathrm{LIS}$ is poly. time

Is P = NP?



Source: `https://en.wikipedia.org/wiki/P_versus_NP_problem`

NP-Completeness



https://en.wikipedia.org/wiki/List_of_NP-complete_problems

If any NP-Hard problem is solvable in poly-time, then every NP problem (1000s of them) is solvable in poly-time.

If any NP-Complete problem cannot be solved in poly-time, then every NP-hard problem (1000s of them) cannot be solved in poly-time.

Problem: Satisfiability (SAT)

Problem

- Given a Boolean formula (or logical expression) in conjunctive normal form (CNF), find either a satisfying truth assignment or report that none exists.
- Examples.
  (i) $(x \vee y \vee z) \wedge (x \vee \overline{y}) \wedge (y \vee \overline{z}) \wedge (z \vee \overline{x}) \wedge (\overline{y} \vee \overline{y} \vee \overline{z})$
  No solution exists.
  (ii) $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$
  Solution is $(x_1, x_2, x_3, x_4) = (\mathsf{T}, \mathsf{T}, \mathsf{F}, \mathsf{F})$
- Applications.
  Circuit design, image analysis, software engineering, artificial intelligence, and automatic theorem proving

Problem: k-Satisfiability (k-SAT)

Problem

- The k-SAT problem is a restricted version of the SAT problem. in which each clause has at most k literals.
- Examples of 3-SAT.
  (i) $(x \vee y \vee z) \wedge (x \vee \overline{y}) \wedge (y \vee \overline{z}) \wedge (z \vee \overline{x}) \wedge (\overline{y} \vee \overline{y} \vee \overline{z})$
  No solution exists.
  (ii) $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$
  Solution is $(x_1, x_2, x_3, x_4) = (\mathsf{T}, \mathsf{T}, \mathsf{F}, \mathsf{F})$

Proving a problem NP-Complete

Proving the first problem SAT is NP-Complete

1. Show that SAT is in NP.
2. Reduce (in poly. time) every problem in NP to SAT.

Proving a new problem X is NP-Complete

1. Show that problem X is in NP.
2. Reduce (in poly. time) an existing NP-Complete problem to X.

3-SAT is in NP-Complete

Problem

- Prove that 3-SAT is NP-Complete.

Solution

1. Show that 3-SAT is in NP:
   Suppose we are given a solution to the 3-CNF Boolean expression.
   In polynomial time we can verify whether the given truth assignment is correct.
2. Show that $\text{SAT} \to 3\text{-SAT}$:
   ?

Reduction: SAT $\to$ 3-SAT

Let $C = (a_1 \vee a_2 \vee A)$ where $A = (a_3 \vee \cdots \vee a_k)$
Let $C' = (a_1 \vee a_2 \vee y) \wedge (\bar{y} \vee A)$

Proof

- [If C is satisfiable, then C' is satisfiable.]
  [Case $a_1 = 1$ or $a_2 = 1$.] $C = 1$.
  Assign $y = 0$ to get $C' = (a_1 \vee a_2 \vee 0) \wedge (1 \vee A) = 1$.
  [Case $A = 1$.] $C = 1$.
  Assign $y = 1$ to get $C' = (a_1 \vee a_2 \vee 1) \wedge (0 \vee A) = 1$.
- [If C' is satisfiable, then C is satisfiable.]
  $C' = 1 \Rightarrow (a_1 \vee a_2 \vee y = 1)$ and $(\bar{y} \vee A) = 1$
  [Case $y = 0$.] Then $(a_1 \vee a_2) = 1 \implies C = 1$.
  [Case $y = 1$.] Then $A = 1 \implies C = 1$.

Reduction: SAT $\rightarrow$ 3-SAT

$\mathrm{SAT}(F)$

1. for each clause C in F with k literals do
2. create $k-3$ tiny clauses of size 3; using a total of $k-3$ new variables; call this collection of tiny clauses C'
3. let the obtained formula be called F'
4. return $3\text{-}\mathrm{SAT}(F')$

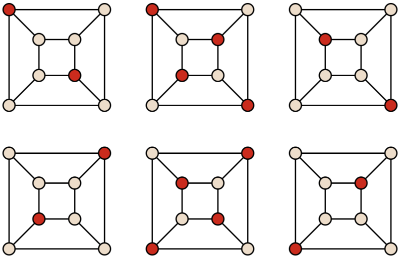
SAT poly. time reduces to $3\text{-}\mathrm{SAT}$

Problem: IndependentSet

An independent set of a graph G is a subset of the vertices such that no two vertices in the subset represent an edge of G.

Problem

- Problem: Independent set (decision version)
  Input: $G = (V, E)$ and an integer k.
  Output: Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?



Source: Wikipedia. Max. independent set size is 4.

IndependentSet is in NP-Complete

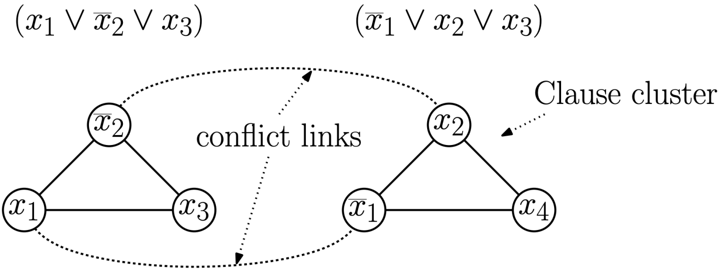
Problem

- Prove that $\textsc{IndependentSet}$ is NP-Complete.

Solution

1. Show that $\textsc{IndependentSet}$ is in NP:
   Suppose we are given a subset S of the vertices in a graph. In polynomial time we can verify that, for each pair of vertices in the set S, there is no edge between them.
2. Show that $3\text{-SAT} \rightarrow \textsc{IndependentSet}$:
   ?

Reduction: 3-SAT $\rightarrow$ IndependentSet



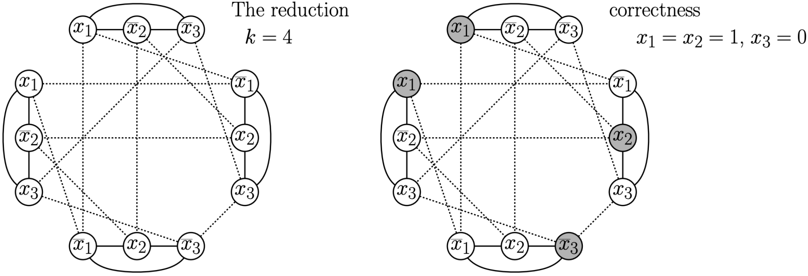
Source: David Mount's notes

Analysis

- A clause (of size at most 3) can be transformed to a clause cluster (of size at most 3)
- Add edges between x_i and all its complement vertices $\bar{x_i}$

Reduction: 3-SAT $\rightarrow$ IndependentSet

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$$



Source: David Mount's notes

Analysis

- Given a k, one needs to select at k vertices that satisfy the independent set property
- Select a vertex from each clause without violating the independent set property

Reduction: 3-SAT $\rightarrow$ IndependentSet

3-SAT(F)

[Transform the input: Boolean expression to graph]

1. $k \leftarrow$ number of clauses in F
2. for each clause $(x_1 \vee x_2 \vee x_3)$ in F do
3. create a clause cluster consisting of three vertices labeled x_1, x_2, x_3
4. create edges $(x_1, x_2), (x_2, x_3), (x_3, x_1)$ between all pairs of vertices in the cluster
5. for each vertex x_i do
6. create edges between x_i and all its complement vertices $\bar{x}_i$ (conflict links)

....

[Transform the output: Vertex set to truth assignment]

7. return $\textsc{IndependentSet}(G, k)$

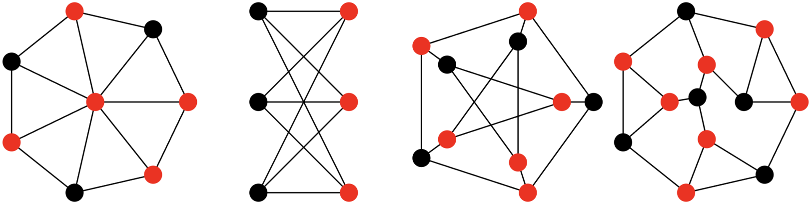
3-SAT poly. time reduces to $\textsc{IndependentSet}$

Problem: VertexCover

A vertex cover of a graph G is a subset of the vertices that touch/cover all edges of G.

Problem

- Problem: Minimum vertex cover (decision version)
  Input: $G = (V, E)$ and a natural number k.
  Output: Check if there exists a set of k vertices that cover all edges.



Source: Mathworld Wolfram.

VertexCover is in NP-Complete

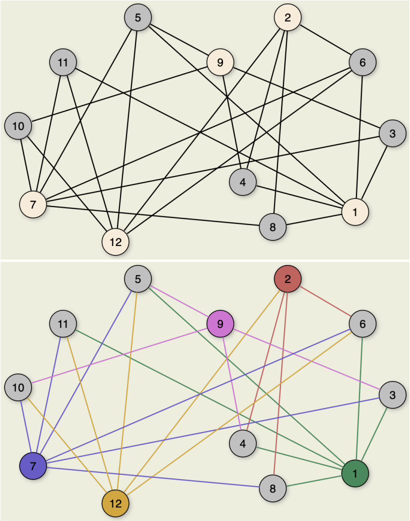
Problem

- Prove that VertexCover is NP-Complete.

Solution

1. Show that VertexCover is in NP:
   Suppose we are given a subset S of the vertices in a graph. In polynomial time we can verify that, for each vertex in the set S, the edges the vertex covers/touches.
2. Show that IndependentSet $\rightarrow$ VertexCover:
   ?

Reduction: IndependentSet → VertexCover



Source: `https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/index.html`

$|V| = 12$. Independent set size $= 7$. Vertex cover size $= 5$.

Reduction: IndependentSet $\to$ VertexCover

In a graph G, S is an independent set $\Leftrightarrow$ $(V-S)$ is a vertex cover

Proof

- [If S is an independent set, then $(V-S)$ is a vertex cover.]
  If S is an independent set, there is no edge $e=(u,v)$ in G, such that both $u,v \in S$. Hence for any edge $e=(u,v)$, at least one of u,v must lie in $(V-S)$.
  $\implies$ $(V-S)$ is a vertex cover in G.
- [If $(V-S)$ is a vertex cover, then S is an independent set.]
  If $(V-S)$ is a vertex cover, between any pair of vertices $(u,v) \in S$ if there exists an edge e, none of the endpoints of e would exist in $(V-S)$ violating the definition of vertex color. Hence, no pair of vertices in S can be connected by an edge.
  $\implies$ S is an independent set in G.

Reduction: IndependentSet $\rightarrow$ VertexCover

$\textsc{IndependentSet}(G, k)$
1. return $\textsc{VertexCover}(G, \lvert V\rvert - k)$

$\textsc{IndependentSet}$ poly. time reduces to $\textsc{VertexCover}$

TSP is in NP-Complete

Problem

- Prove that TSP is in NP-Complete.

Solution

1. Show that TSP is in NP:
   Suppose we are given a tour in a graph and a natural number k.
   In polynomial time we can verify if the given solution is really a tour (covers each vertex exactly once, except the last vertex) and if the total weight of the tour is less than or equal to k.
2. Show that $\textsc{HamiltonianCycle} \rightarrow \mathrm{TSP}$:
   ?

Reduction: HamiltonianCycle $\rightarrow$ TSP

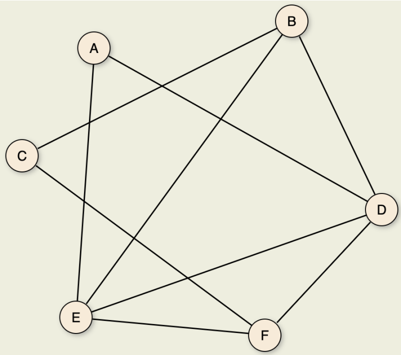
Problem

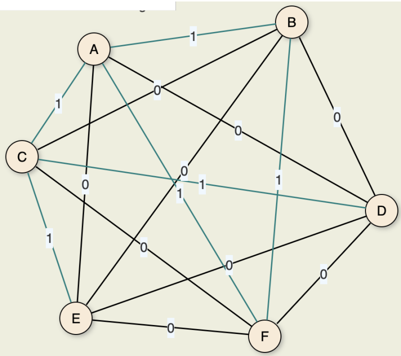
- Problem: Hamiltonian cycle
  Input: $G = (V, E)$.
  Output: Check if the graph contains a Hamiltonian cycle, i.e., a cycle that passes through all the vertices of the graph exactly once.

Problem

- Problem: TSP
  Input: Weighted graph $G = (V, E)$ with nonnegative weights and a natural number k.
  Output: Check if the graph contains a simple cycle of length $\leq k$ (i.e., total weight cost) that passes through all the vertices of the graph exactly once.

Reduction: HamiltonianCycle $\rightarrow$ TSP





Source: `https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/index.html`

Hamiltonian cycle in first graph $\Leftrightarrow$ finding TSP of cost 0 is second graph.

Reduction: HamiltonianCycle $\rightarrow$ TSP

Transformation

For a given graph $G = (V, E)$, create the graph $G' = (V', E')$ as follows:

- [vertices.] $V' = V$
- [edges.] $E' = \{(u, v)\}$ for unique vertices u, v in V'
- [weights.] for each edge e in E':
  $w(e) = 0$ if e is in E,
  $w(e) = 1$ if e is not in E

Reduction: HamiltonianCycle $\rightarrow$ TSP

$\textsc{HamiltonianCycle}(G) \Leftrightarrow \text{TSP}(G', 0)$

Reduction

- $[\textsc{HamiltonianCycle}(G) \Rightarrow \text{TSP}(G', 0).]$
  If G contains a Hamiltonian cycle, it forms a cycle in G' with total cost 0 because the weights of all the edges is 0. Hence, there exists a TSP solution in G' with total cost ≤ 0.
- $[\text{TSP}(G', 0) \Rightarrow \textsc{HamiltonianCycle}(G).]$
  If G' contains a cycle that passes through all vertices exactly once, and has length ≤ 0, then the cycle contains only the edges that were originally present in graph G. Hence, there exists a Hamiltonian cycle in G.

Reduction: HamiltonianCycle $\rightarrow$ TSP

HamiltonianCycle$(G = (V, E))$

1. construct complete graph $G' = (V', E')$ such that $V' = V$
2. for each edge e in E' do
3. if e is in E then
4. $w(e) \leftarrow 0$
5. else
6. $w(e) \leftarrow 1$
7. return $\mathrm{TSP}(G', 0)$

HamiltonianCycle poly. time reduces to TSP