The Theory of NP

Tractable and intractable problems
NP, NP-complete & NP-hard problems

The theory of NP-completeness

- Tractable and intractable problems
- NP-complete problems

Classifying problems

- Classify problems as tractable or intractable.
- Problem is *tractable* if there **exists at least one** polynomial bound algorithm that solves it
- An algorithm is *polynomial bound* if its worst case time complexity is bounded by a polynomial *p*(*n*) in the size *n* of the problem

$$p(n) = a_n n^k + \ldots + a_1 n + a_0 \text{ where } k \text{ is a constant}$$

Intractable problems

- Problem is *intractable* if it is not tractable.
- **1st Category: All** algorithms that solve the problem are not polynomial bound.
- It has a worst case growth rate *f*(*n*) which cannot be bound by a polynomial p(n) in the size n of the problem.
- For intractable problems the bounds are:

$$f(n) = c^n \text{ , or } n^{\log n}\text{, etc.}$$

Another set of intractable problems

- 2^{nd} **category**: Undecidable problems
  - Cannot give a “yes” or “no” answer
  - E.g., **Halting problem**
  - *No algorithm can be devised to solve the halting problem*

Halting problem

- Input: A string P and a string I. Consider P as a program and I as input to P.
- Output: 1 if P halts on I; 0 if P does not halt on I (infinite loop)
- **Theorem (Turing circa 1940): There is no program to solve the halting problem.** See next slide for proof.

Proof: Halting problem is undecidable

- Proof: To reach a contradiction, assume that there exists a program Halt(P, I) that solves the halting problem. Halt(P, I) returns true if and only if P halts on I. Otherwise, it returns false. Using Halt(P, I), we construct the following program Z:

```
program (string x)
begin
    If Halt (x, x) then
        while(1) printf ("ha ha ha ");
    Else exit(0)
end
```

- Case 1: Program Z halts on input Z. By the correctness of Halt, Halt(Z, Z) returns true. Thus, program Z loops forever on input Z, printing “ha ha ha ….” Contradiction.

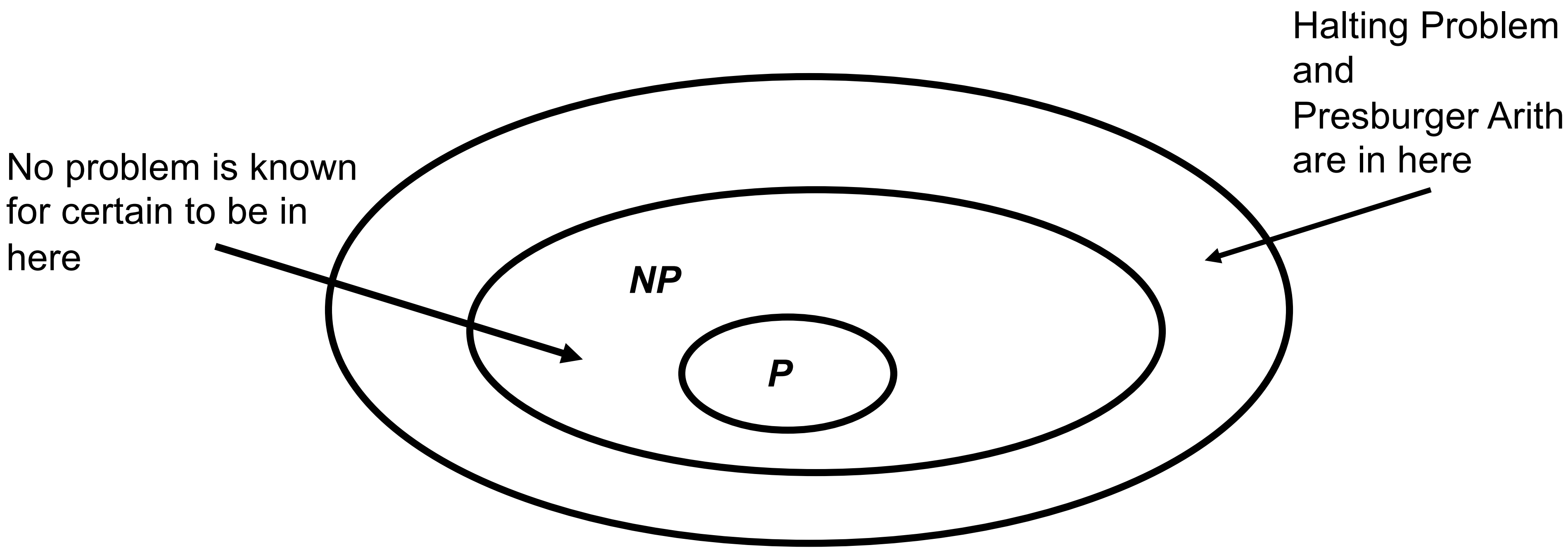
- Case 2: Program Z does not halt on input Z. Halt(Z, Z) returns false. Hence, program Z halts. Contradiction.

Why is this classification useful?

- If problem is intractable, no point in trying to find an *efficient* algorithm that solves the problem with polynomial time complexity in the worst case
- All algorithms will be too slow for **large inputs**.

Intractable problems

- Turing showed some problems are so hard that no algorithm can solve them (undecidable)
- Other researchers showed some decidable problems from automata, mathematical logic, etc. are intractable: Presburger arithmetic is doubly exponential

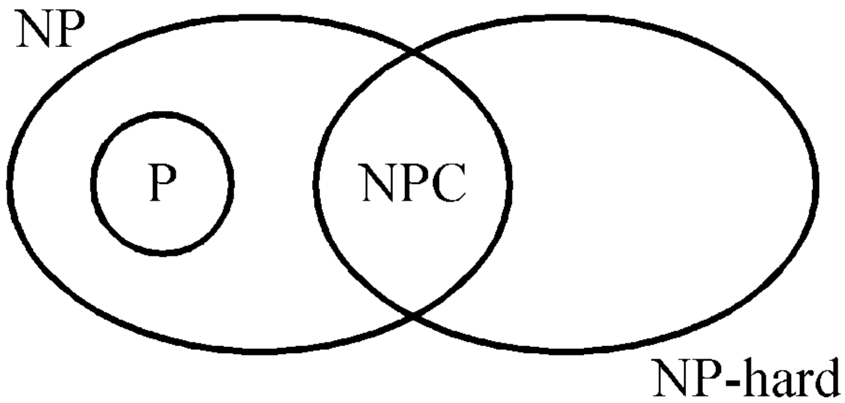


Problems Proven to be Intractable

- All Hamiltonian circuits: For a complete undirected graph, there are (n-1)! Circuits
- Halting problem: Undecidable
- Presburger Arithmetic
- …

Problems not proven to be intractable but no poly. time alg.

- 0-1knapsack
- Traveling salesperson
- Sum of subsets
- M-coloring for $m \geq 3$
- …



- **NP** : the class of problem which can be solved by a non-deterministic polynomial algorithm.
- **P**: the class of problems which can be solved by a deterministic polynomial algorithm.
- **NP-hard**: the class of problems to which every NP problem reduces.
- **NP-complete (NPC)**: the class of problems which are NP-hard and belong to NP.

Coping with NP-Complete/NP-Hard Problems

- Rely on approximation algorithms, heuristics, etc.

- Sometimes we need to solve only a restricted version of the problem.

- If the restricted problem is tractable, design an algorithm for the restricted version

Nondeterministic algorithms

- A nondeterminstic algorithm consists of
  phase 1: guessing
  phase 2: checking
- If the checking stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (**nondeterministic polynomial**) algorithm.
- NP problems: (must be decision problems)
  - e.g. searching, MST, sorting
    satisfiability problem (SAT)
    traveling salesperson problem (TSP)

Nondeterministic operations and functions

- Choice(S) : arbitrarily chooses one of the elements in set S
- Failure : an unsuccessful completion
- Success : a successful completion
- Nonderministic searching algorithm:

```
  j ← choice(1 : n)   /* guessing */
 if A(j) = x then success  /* checking */
              else failure
```

- A nondeterministic algorithm terminates unsuccessfully iff there exist no set of choices leading to a success signal.
- The time required for *choice(1 : n)* is O(1)

Hard practical problems

- There are many practical problems for which *no one has yet* found a polynomial bound algorithm.
- Examples: 3-SAT, traveling salesperson, 0/1 knapsack, sum of subsets, graph coloring, bin packing etc.
- Most design automation problems such as testing and routing.
- Many OS, networks, database and graph problems.

Satisfiability (SAT) problem

Conjunctive Normal Form (CNF)

- A literal is a variable or the negation of a var.
  - Example: The variable x is a literal, and its negation, $\neg x$, is a literal.
- A clause is a disjunction (an OR) of literals.
  - Example: $(x \vee y \vee \neg z)$ is a clause
- A formula is in Conjunctive Normal Form (CNF) if it is a conjunction (an AND) of clauses.
  - Example: $(x \vee \neg z) \wedge (y \vee z)$ is in CNF.
- A CNF formula is a conjunction of disjunctions, i.e., a product (AND) of sums (OR)

Definition: A CNF formula is a 3CNF-formula iff each clause has exactly 3 literals.

$$\underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{\text{clauses}} \wedge \underbrace{(x_4 \vee x_2 \vee x_5)}_{\text{clauses}} \wedge \ldots \wedge \underbrace{(x_3 \vee \neg x_2 \vee \neg x_1)}_{\text{clauses}}$$

- **A literal is a variable or the negation of a var.**
- **A clause is a disjunction (an OR) of literals.**
- **A formula is in Conjunctive Normal Form (CNF) if it is a conjunction (an AND) of clauses.**
- **A CNF formula is a conjunction of disjunctions of literals.**

Definition: A CNF formula is a 3CNF-formula iff each clause has exactly 3 literals.

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_2 \vee x_5) \wedge \ldots \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$$

clauses

YES $(x_1 \vee \neg x_2 \vee x_1)$

NO $(x_3 \vee x_1) \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$

NO $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_4 \vee x_2 \vee x_1) \vee (x_3 \vee x_1 \vee \neg x_1)$

NO $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_3 \wedge \neg x_2 \wedge \neg x_1)$

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

Boolean Basics: Literals, Clauses, CNF

- Boolean function on n variables is a mapping $\{0,1\}^n \rightarrow \{0,1\}$
- Literal = Boolean variable or its negation
- Clause = disjunction of literals (no complementary pair)
- Conjunctive Normal Form (CNF) = conjunction of clauses, i.e., product-of-sums (<u>Fact</u>: Every Boolean function has a CNF representation)

Cook's theorem

- SAT is NP-complete
- 3-SAT is NP-complete (1-SAT or 2-SAT is P)
- It is the first NP-complete problem
- Every NP problem reduces to SAT
- NP = P iff the SAT problem is a P problem

How are they handled?

- A variety of algorithms based on backtracking, branch and bound, dynamic programming, etc.
- None can be shown to be polynomial bound (exponential in the worst case)

Theory of NP completeness

- The theory of NP-completeness enables showing that these problems are at least as hard as *NP-complete* problems
- Practical implication of knowing a problem is NP-complete is that it is **probably** intractable (whether it is or not has not been proved yet)
- So any algorithm that solves it will probably be very slow for large inputs

We will need to discuss

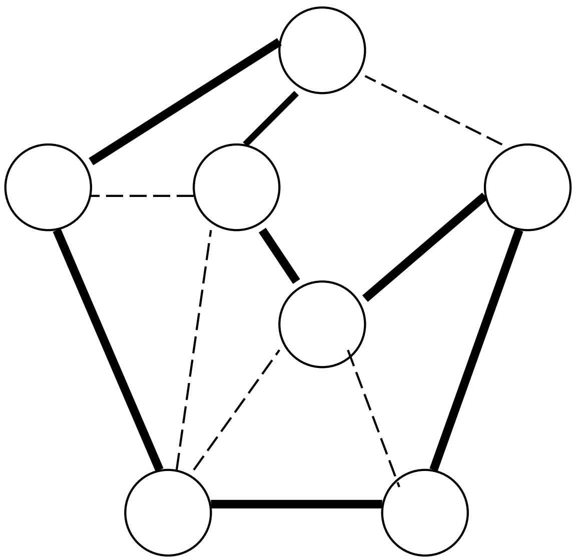
- Decision problems
- Converting optimization problems into decision problems
- The relationship between an optimization problem and its decision version
- The class P
- Verification algorithms
- The class NP
- The concept of polynomial transformations
- The class of NP-complete problems

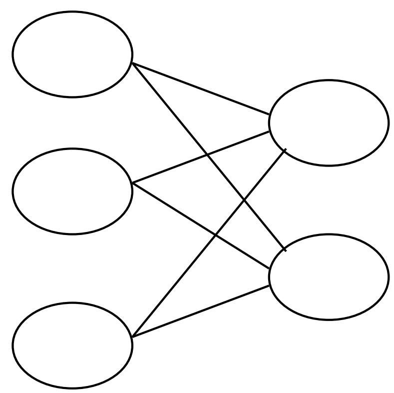
Decision Problems

- A *decision* problem answers *yes or no* for a given input
- Examples:
  - Given a graph *G,* is there a path from *s* to *t* of length at most *k*?
  - Does graph *G* contain a Hamiltonian cycle?
  - Given a graph *G,* is it bipartite?
  - For a 0-1 knapsack problem, is there a solution whose benefit is $100 or more?

A decision problem: HAMILTONIAN-CYCLE

- A *Hamiltonian cycle* of a graph *G* is a cycle that visits each vertex of the graph (except for the starting node) exactly once.
- Problem: Given a graph G, does G have a Hamiltonian cycle?





Converting to decision problems

- Optimization problems can be converted to decision problems (typically) by adding a bound B on the value to optimize, and asking the question:
  - Is there a solution whose value is at most B? (for a minimization problem)
  - Is there a solution whose value is at least B? (for a maximization problem)

An optimization problem: traveling salesman

- Given:
  - A finite set $C = \{c_1, \ldots, c_m\}$ of cities and
  - A distance function $d(c_i, c_j)$ of nonnegative numbers
- Find the length of the **minimum** distance tour which visits every city exactly once and comes back to the starting city

A decision problem for traveling salesman

- Given a finite set $C = \{c_1, \ldots, c_m\}$ of cities, a distance function $d(c_i, c_j)$ of nonnegative numbers and a bound B

- Is there a tour of all the cities (in which each city is visited exactly once) with total length **at most B**?

- There is no known polynomial bound algorithm for TS.

Relation between an optimization problem and the decision problem

- If we have a solution to the optimization problem we can compare the solution to the bound and answer "yes" or "no"
- Therefore if the optimization problem is tractable so is the decision problem
- If the decision problem is "hard" the optimization problem is also "hard"
  - If the optimization is easy then the decision problem is easy

The class P

- **P is the class of decision problems that are polynomial bound**

- Is the following problem in P?
  - Given a weighted graph G, is there a spanning tree of weight at most B?

- The decision versions of problems such as shortest distance path and minimum spanning tree belong to P
  - Simply compute an MST and find its weight to B

The goal of verification algorithms

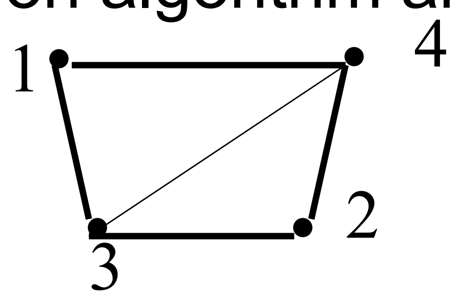
- The goal of a verification algorithm is to verify a “yes” answer to a decision problem’s input (i.e., if the answer is “yes” the verification algorithm verifies this answer)

- The inputs to the verification algorithm are:
  - the original input (problem instance) and
  - a *certificate* (possible solution)

Verification Algorithms

- A *verification algorithm* takes a problem instance x and *answers "yes"*, if there **exists** a certificate y such that the answer for x with certificate y is "yes"

- Consider HAMILTONIAN-CYCLE
- A problem *instance* x lists the vertices and edges of G: ({1,2,3,4}, {(3,2), (2,4), (3,4), (4,1), (1, 3)})
- There **exists** a certificate y = (3, 2, 4, 1, 3) for which the verification algorithm answers "yes"



Polynomial bound verification algorithms

- Given a decision problem d

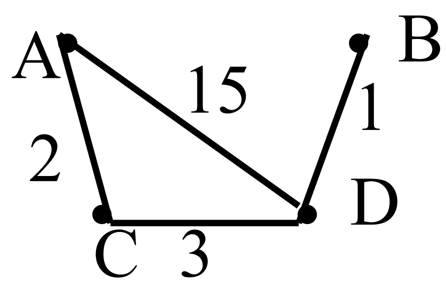
- A verification algorithm for d is *polynomial bound* if given an input x to d, there exists a certificate y, such that $|y|=O(|x|^c)$ where c is a constant, and a polynomial bound algorithm A(x, y) that verifies an answer "yes" for d with input x

Note: |y| is the size of the certificate, |x| is the size of the input

The problem PATH

- PATH denotes the decision problem version of shortest path.
- PATH: Given a graph *G*, a start vertex *u*, and an end vertex *v*. Does there exist a path in *G*, from *u* to *v* of length at most *k?*

- The instance is: G=({A, B, C, D}, {(A, C,2), (A, D, 15), (C,D, 3), (D, B, 1)} k=6
- A certificate y=(A, C, D, B)



A verification algorithm for PATH

- Verification algorithm:
  - Given the problem instance x and a certificate *y*
    - Check that *y* is indeed a path from *u* to *v*.
    - Verify that the length of *y* is at most *k*

- Is the verification algorithm for PATH polynomial bound?
- Is the size of y polynomial in the size of x?
- Is the verification algorithm polynomial bound?

Example: A verification algorithm for TS (Traveling Salesman)

- Given a problem instance x for TS and a certificate *y*
  - Check that *y* is indeed a cycle that includes every vertex exactly once except for the starting node
  - Verify that the length of the cycle is at most B

- Is the size of y polynomial in the size of x?
- Is the verification algorithm polynomial?

The class NP (Nondeterministic Polynomial)

- **NP is the class of decision problems for which there is a polynomial bound verification algorithm**

- It can be shown that:
  - all decision problems in P, and
  - decision problems such as traveling salesman, knapsack, bin packing, are also in NP

The relation between P and NP

- P $\subseteq$ NP
- It is not known whether P = NP or P $\neq$ NP
- Problems in P can be *solved* "quickly"
- Problems in NP can be *verified* "quickly"
- It is easier to verify a solution than solving a problem
- Some researchers believe that P and NP are not the same class (But no one has proved whether or not this is true)

Polynomial reductions

- **Motivation**: The definition of NP-completeness uses the notion of *polynomial reductions* of one problem A to another problem B, written as

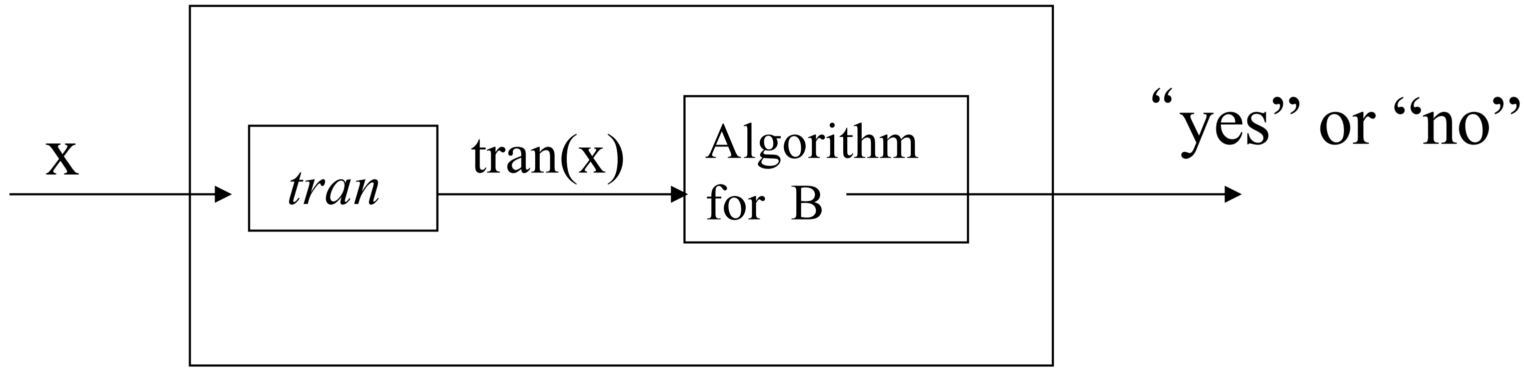
$$A \propto B$$

- Let *tran* be a function that converts any input x for decision problem A into input tran(x) for decision problem B

Polynomial reductions

tran is a *polynomial reduction* from A to B if:

1. *tran* can be computed in polynomial bound time
2. The answer to A for input x is *yes* if and only if the answer to B for input *tran*(x) is *yes*.



Algorithm for A

Two simple problems

- A: Given n Boolean variables with values $x_1,\ldots,x_n$, does at least one variable have the value True?
- B: Given n integers $i_1,\ldots,i_n$ is $max\{i_1,\ldots,i_n\}>0$?

Algorithm for B :

Check the integers one after the other.
If one is positive, stop and answer "*yes*"
If none is positive, stop and answer "*no*".

Example:
n=4.
Given integers: -1, 0, 3, and 20.
Algorithm for B answers "*yes*".
Given integers: -1, 0, 0, and 0.
Algorithm for B answers "*no*".

Is there a transformation?

- Can we transform an instance of A into an instance of B?
- Yes.

```
tran(x)
    for ( j =1; j =< n; j ++)
    if (x_j == true)
        i_j=1
    else   // x_j=false
        i_j = 0
```

T(false, false, true, false)= 0,0,1,0

- Is this transformation polynomial bound? yes

Does it satisfy all the requirements?

- Can we show that when the answer for an instance $x_1,\ldots,x_n$ of A is "*yes*" the answer for the transformed instance *tran*$(x_1,\ldots,x_n)= i_1,\ldots,i_n$ of B is also "*yes*"?

- If the answer for the given instance $x_1,\ldots,x_n$ of A is "*yes*", there is some x_j=true.

- The transformation assigns $i_j=1$.

- Therefore the answer for problem B is also "*yes*" (the maximum is positive)

The other direction

- Can we also show that when the answer for problem B with input *tran*($x_1,\ldots,x_n$)= $i_1,\ldots,i_n$ is "*yes*", the answer for the instance $x_1,\ldots,x_n$ of A is also "*yes*"?
- If the answer for problem B is "*yes*", it means that there is an $i_j>0$ in the transformed instance.
- i_j is either 0 or 1 in the transformed instance. If $i_j=1$, x_j=true.
- So the answer for A is also "*yes*"

Polynomial reductions

Theorem:

If $A \propto B$ and B is in P, then A is in P

If A is not in P then B is also not in P

NP-complete problems

- **<u>A problem A is *NP-complete* if</u>**
  <u>1. It is in NP and</u>
  <u>2. For every other problem A' in NP, A'$\propto$ A</u>

- <u>A problem A is ***NP-hard*** if</u>
  <u>For every other problem A' in NP, A'$\propto$ A</u>

$$NP-complete \subseteq NP-hard$$

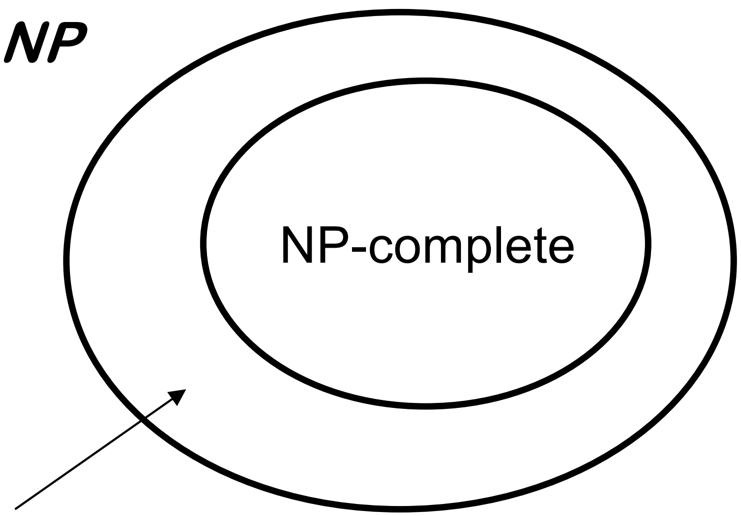
- Example: Halting problem is NP-hard but not NP-complete

Why is NP-complete important?

If any NP-complete problem is in P, then P = NP.

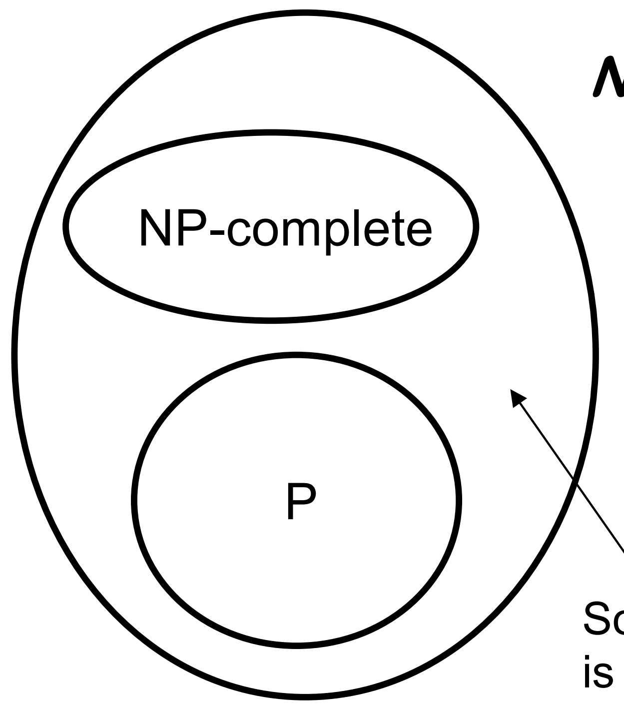
If any NP-complete problem is not polynomial bound, then all NP-Complete problems are not polynomial bound.

P* = *NP



The trivial decision problem
that always answers
“yes” in here

NP* ≠ *P



Some Problem
is in here

NP-completeness and Reducibility

- The existence of NP-complete problems leads us to *suspect* that P $\neq$NP.
- If HAMILTONIAN CYCLE, which is an NP-complete problem, can be solved in polynomial time, every problem in NP can be solved in polynomial time. This means every problem in NP is polynomial bound and, therefore, P=NP.
- If HAMILTONIAN CYCLE could not be solved in polynomial time, every NP-complete problem cannot be solved in polynomial time. Thus **NP** $\neq$ **P**

Revisit the SAT problem

- First, Conjunctive Normal Form (CNF) will be defined
- Second, satisfiability (SAT) problem will be defined
- Finally, we will show a polynomial bounded verification algorithm for the problem

Conjunctive Normal Form (CNF)

- A *logical (Boolean) variable* is a variable that may be assigned the value *true* or *false* (p, q, r and s are Boolean variables)

- A *literal* is a logical variable or the negation of a logical variable (p and $\neg$q are literals)

- A *clause* is a disjunction of literals
  (($p \vee q \vee s$) and ($\neg q \vee r$) are clauses)

Conjunctive Normal Form (CNF)

- A logical (Boolean) expression is in CNF if it is a conjunction of *clauses*
- The following expression is in conjunctive normal form:

$$(p \vee q \vee s) \wedge (\neg q \vee r) \wedge (\neg p \vee r) \wedge (\neg r \vee s) \wedge (\neg p \vee \neg s \vee \neg q)$$

Satisfiability (SAT) problem

- Is there a truth assignment to the n variables of a logical expression in CNF which makes the value of the expression true?
- The answer is yes, if all clauses evaluate to true
- Otherwise, the answer is “no”

SAT problem

- p=T, q=F, r=T and s=T is a truth assignment for:

$(p \vee q \vee s) \wedge (\neg q \vee r) \wedge (\neg p \vee r) \wedge (\neg r \vee s) \wedge (\neg p \vee \neg s \vee \neg q)$

- Note that if q=F then $\neg$q=T

- Each clause evaluates to true

A verification algorithm for SAT

1. Check that the certificate s is a string of exactly n characters which are T or F.
2. **while** (there are unchecked clauses) {
   select next clause
   if (clause evaluates to false) **return**(“no”) }
3. **return** (“yes”)

- Is verification algorithm polynomial bound?
- Satisfiability is in NP since there exists a polynomial bound verification algorithm for it

Cook's theorem

- **SAT (at least 3-SAT) problem is NP complete**
  - Cook proved that SAT is NP and every problem in NP reduces to SAT
  - First problem proved to be NP complete
  - Proof idea: encode the workings of a Nondeterministic Turing machine for an instance *I* of problem $X \in NP$ as a SAT formula so that the formula is satisfiable iff the nondeterministic Turing machine accepts the instance *I*

- After Cook's theorem, many NP-complete problems are found
  - E.g., 3-SAT $\propto$ Hamiltonian Cycle Decision Problem, SAT $\propto$ 3-coloring, 3-SAT $\propto$ Clique, …
  - How to do this? See the following slides

- More NP-complete problems are found from NP complete problems that are not 3-SAT
  - E.g., Hamiltonian cycle $\propto$ Traveling Salesperson, Clique $\propto$ vertex cover …

Shortcut for NP-completeness Proofs

- To prove a language *L* is NP-complete:

Prove L $\in$ NP.

Choose L' $\in$ NPC, and show L' $\propto$ L

- L $\in$ NP. We will show that every M $\in$ NP satisfies M $\propto$ L, and thus L is NP-complete
  - Let M $\in$ NP. M $\propto$ L' (definition of NPC), and L' $\propto$ L (proved by us). So by transitivity M $\propto$ L

Reductions

For example, let's discuss how to:

- Reduce 3-SAT to Clique
- Reduce Clique to Vertex Cover
- Reduce 3-SAT to Hamiltonian Cycle
- Reduce Hamiltonian Cycle to TSP

Clique

- Show clique is a NP-complete problem via reduction

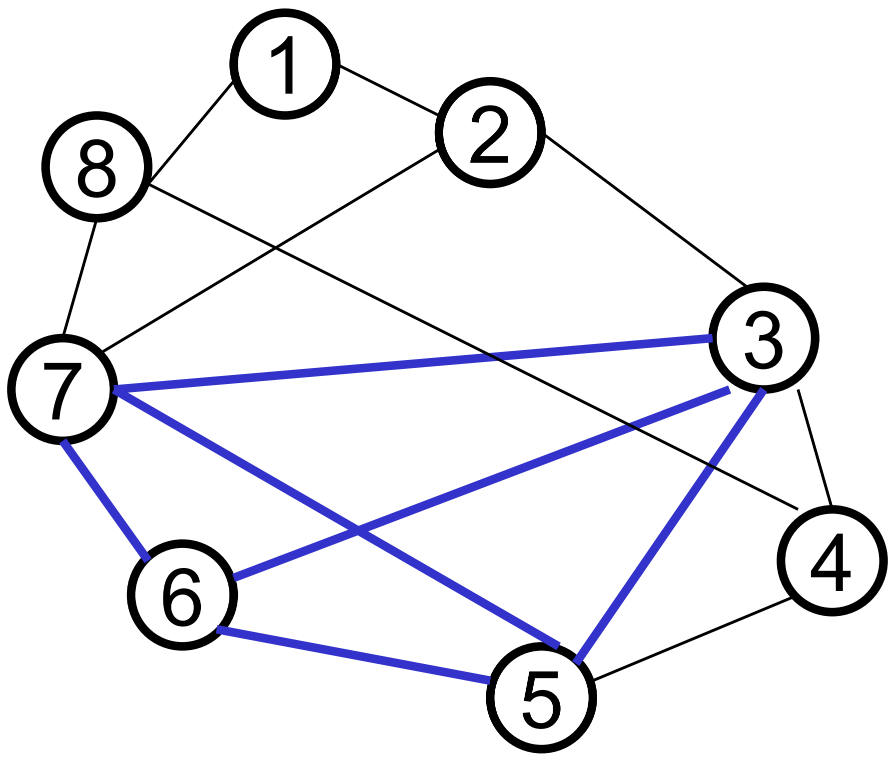
The Clique Problem

- **A *clique* is a complete undirected graph where every vertex is connected to every other vertex.**

CLIQUE

- **Input**: **An undirected graph *G* and a positive integer *k*.**
- **Output**: **YES iff a clique of size *k* exists in *G*.**

Clique example



- G contains a clique of 4 (with vertices 3, 5, 6, 7)
- The 4 people 3, 5, 6, 7 “know” (can work with each other) each other

The Clique Problem

- **Theorem:** CLIQUE is NP-complete.
- **Proof:**
- **Step 1.** CLIQUE $\in$ NP

  Given a certificate that contains a set of k vertices $V' \subseteq V$, we can check if V' forms a clique by checking for every pair of nodes u, v $\in$V' that (u,v) $\in$ E

- Clearly, this can be done in polynomial time.

The Reduction

Step 2. Selection

3-CNF-SAT which is NP-Complete.

Step 3. Mapping

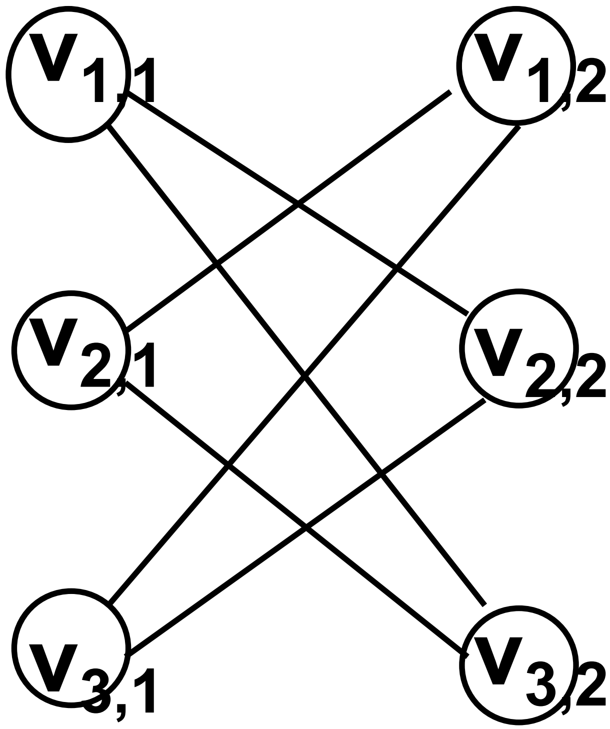
For a formula $C_1 \wedge \ldots \wedge C_k$ such that $C_r = l_{1,r} \vee l_{2,r} \vee l_{3,r}$ we construct a graph G with vertices $v_{1,r}\ v_{2,r}\ v_{3,r}$ for $r = 1, \ldots, k$, where $v_{i,r}$ represents the literal $l_{i,r}$

The Reduction

We put an edge between $v_{i,r}$ and $v_{j,s}$ if both of the following hold:

1. $r \neq s$ and
2. $l_{i,r}$ is not the negation of $l_{j,s}$.

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



k=2
The graph has 6 cliques of size 2

Step 4a. Yes for 3-Sat implies yes for clique

- **Assume formula satisfiable.**
- **With the satisfying assignment each clause contains at least 1 literal that is assigned 1.**
- **Since each literal from each clause is a vertex in the graph, if we pick out a literal that is assigned 1 from each of the k clauses, we get k vertices in the graph.**

Step 4a. Yes for 3-SAT implies yes for Clique

- **This set of k vertices is a clique.**
  - **For any two vertices, the corresponding literals are from different clauses, and are both assigned 1, so they cannot be complements of a single variable**
  - **Thus there is an edge between any two such vertices.**

Step 4b. Yes for Clique implies yes for 3-Sat

- **Assume G has a clique V' of size k**

- **No edge connects vertices in the same clause, so each of k triples has exactly one vertex in V'**

- **Assign 1 to each literal in V' without getting an inconsistent assignment (why?), and assign arbitrary values to the rest of the variables**

- **For this assignment, each clause is satisfied and thus the answer for 3-SAT is yes**

Step 5. Reduction is polynomial

- **Step 5. The reduction is polynomial.**
  - **The formula is read and 3k vertices are generated in O(k) steps. Then, each pair of literals ($9\binom{k}{2}$) from two different clauses is checked and an edge is added if the literals are not complimentary.**
  - **The reduction is $O(k^2)$**

Vertex Cover

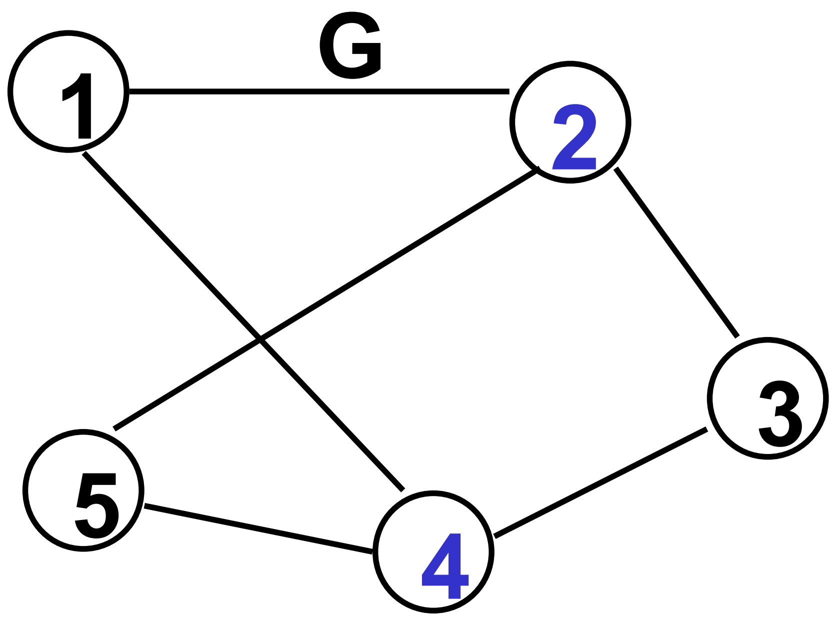
- Reduce clique to vertex cover

The vertex-cover problem

- **A *vertex cover* of an undirected graph is a set of vertices V' such that for every edge (u,v), either u or v or both are in V'. The problem is to find a cover of minimum size.**

- **VERTEX-COVER**
  - **Input: A graph G and a number k.**
  - **Output: YES iff G has a vertex cover of size k.**

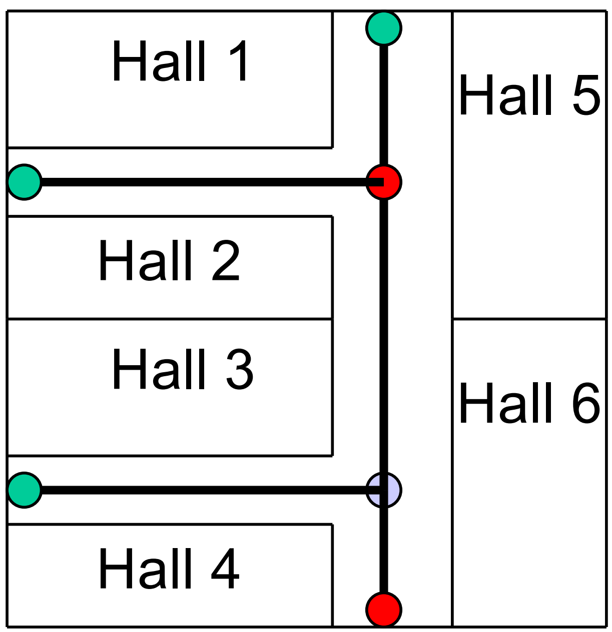
Example of a vertex cover problem

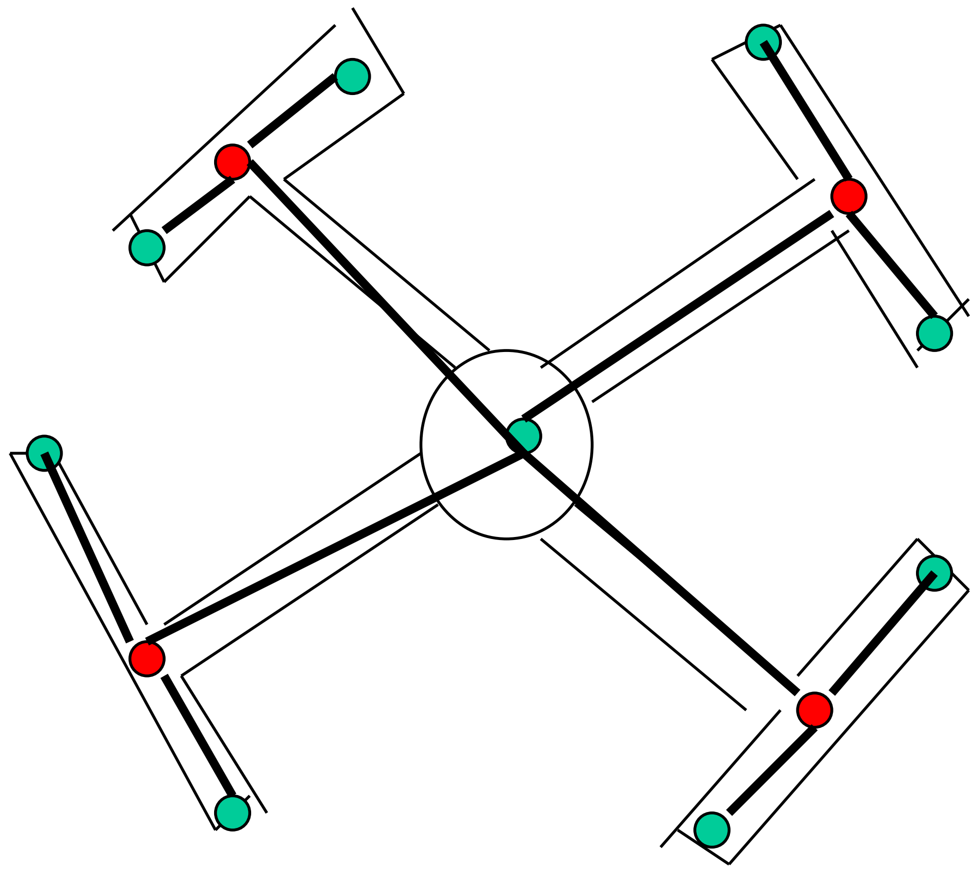


k=2

Application of vertex cover

- What is the fewest # of guards we need to place in a museum to cover all the corridors? An airport to cover all the main walkways





The vertex-cover problem

- **Theorem**: **VERTEX-COVER is NP-complete.**

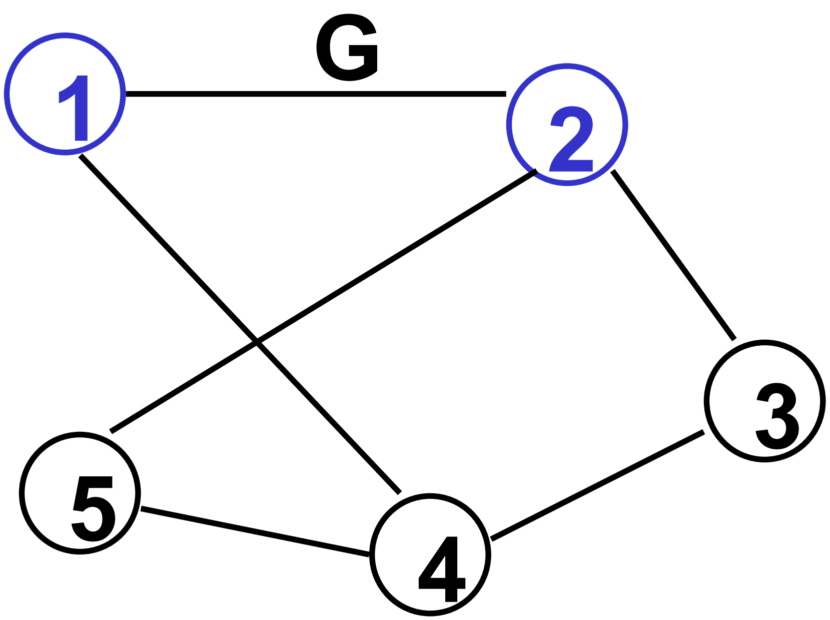
- **Proof**: **Step 1**. **VERTEX-COVER $\in$ NP (obvious algorithm, given a subset of vertices).**

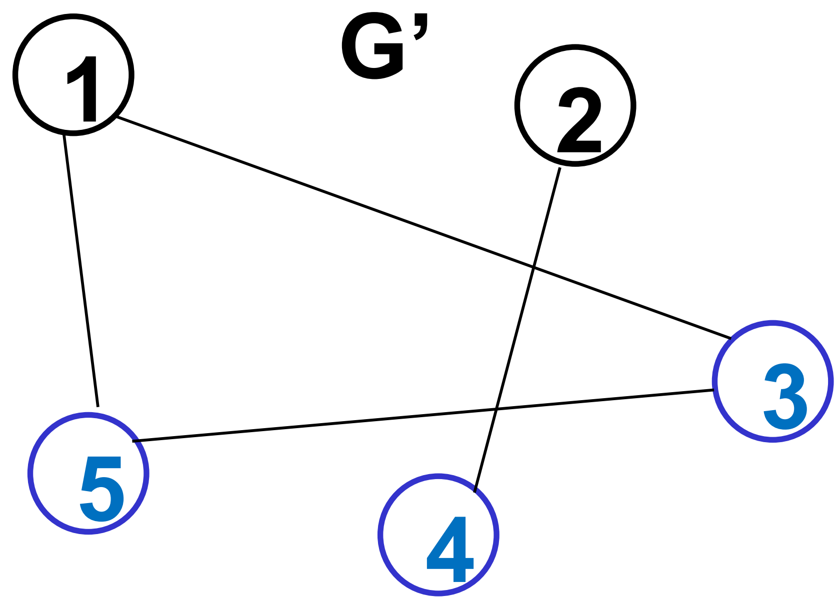
- **Step 2**. **We select CLIQUE (will show that CLIQUE $\propto$ VERTEX-COVER)**

The reduction

- **Step 3. The mapping.**
- **Given an instance of the CLIQUE problem <G, k> we output an instance <G', |V|-k> of the VERTEX-COVER problem.**
- **G' has the same vertices as G and exactly those edges that are not in G.**
- **It is easy to show the reduction is polynomial (step 5)**

Reduction Example



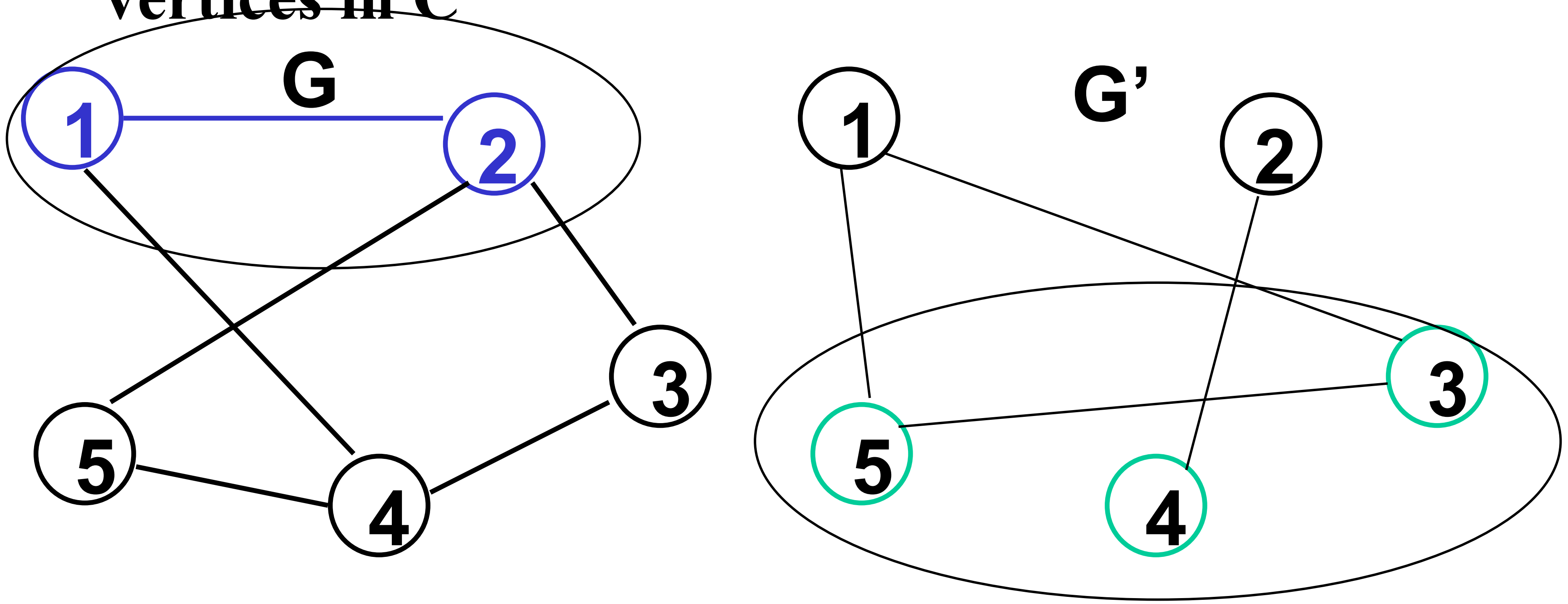


Clique {1,2} of size 2

Cover {3,4,5} of size 3

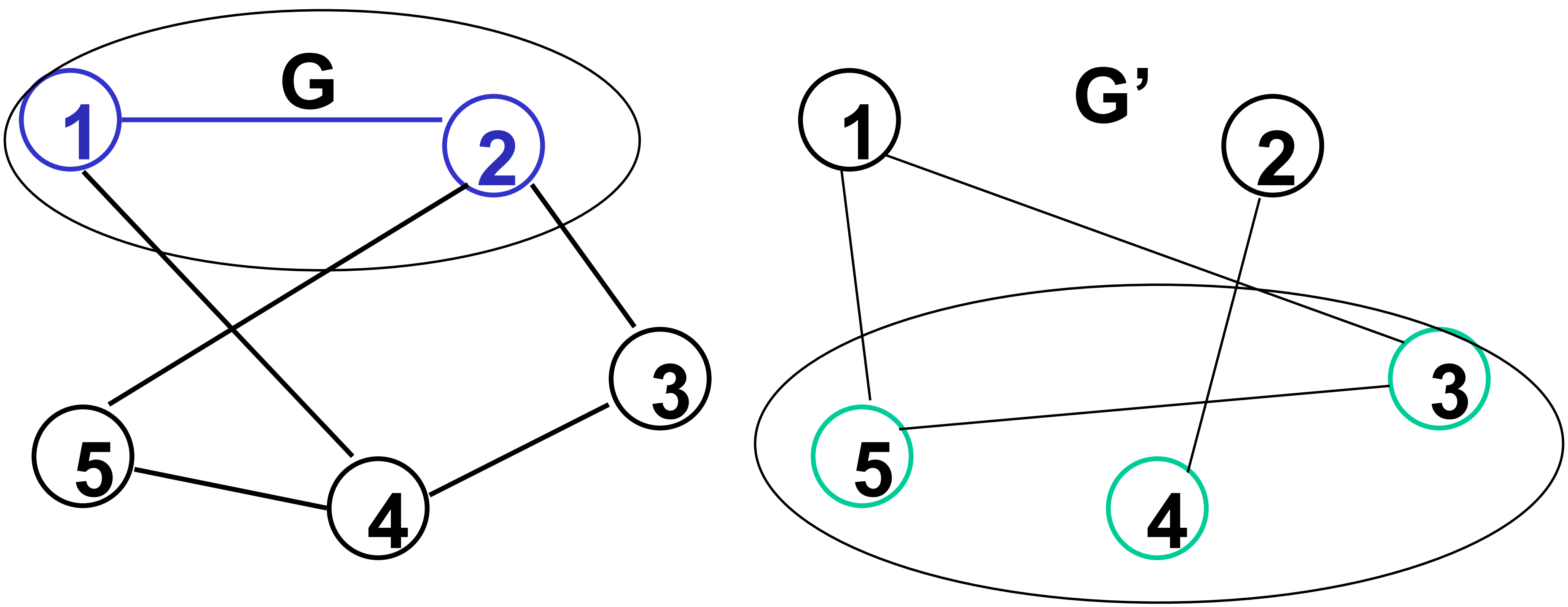
Step 4. Correctness of the reduction

- **Assume G has a clique C of size k.**
- **In G' there are no edges between any pair of vertices in C**



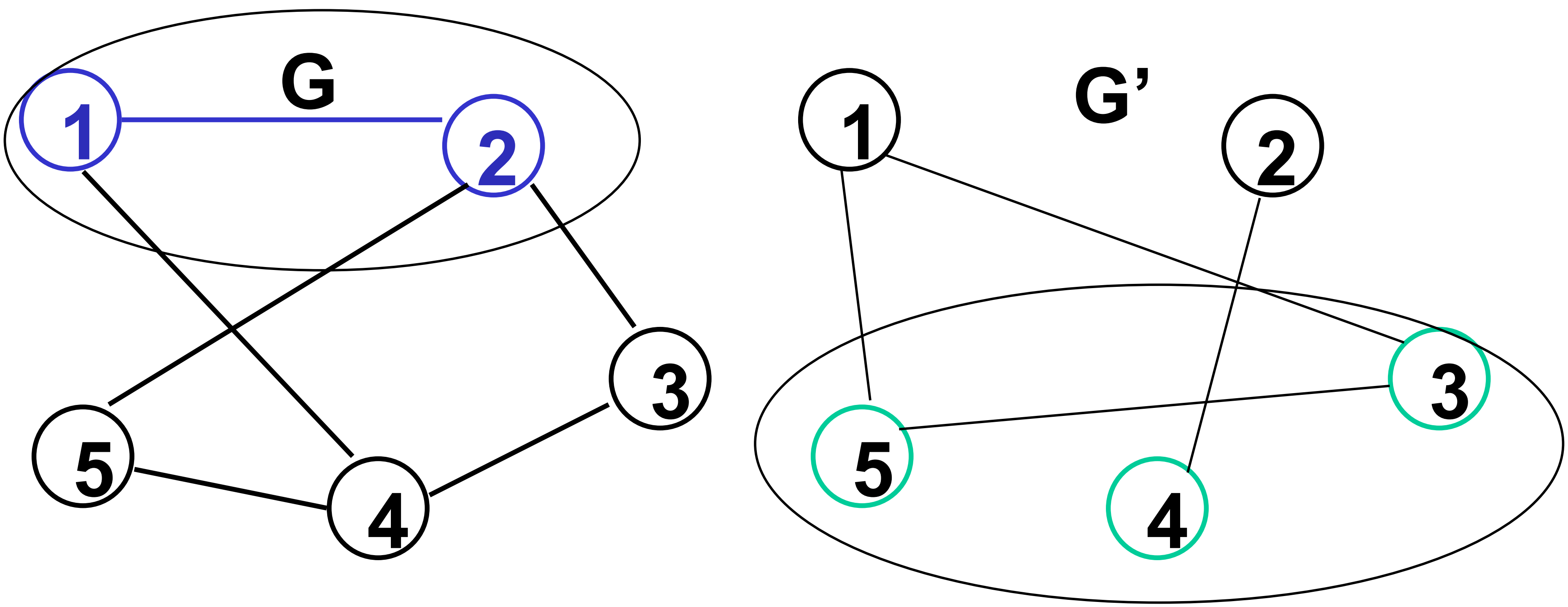
Step 4 cont

- So all edges in G' are between a node in C and a node in V-C, or two nodes in V-C.
- So V-C is a vertex cover for G'.



Step 4. Correctness of the reduction

- Assume G'=(V, E') has a vertex cover $V' \subseteq V$, where $|V'| = |V|-k$.
- Thus for all $u, v \in V-V'$ (not in the cover), $(u,v) \notin E'$ and thus $(u,v) \in E$
- V-V' is thus a clique.



Hamiltonian Cycle

- **A *Hamiltonian cycle* of a graph G is a cycle that contains each vertex in V exactly once. A graph is *Hamiltonian* if it has a Hamiltonian cycle.**

- **HAM-CYCLE**
  - **Input: A graph G.**
  - **Output: YES iff G is Hamiltonian.**

- **Theorem: HAM-CYCLE is NP-complete.**
  - **3-CNF-SAT $\propto$ HAM-CYCLE (proof omitted).**

Traveling Salesperson

- Reduce Hamiltonian Cycle to Traveling Salesperson

Traveling Salesman

- A *tour* is a Hamiltonian cycle in a graph. We want the minimum cost tour in a weighted graph.

- **TSP:**
  - **Input**: A graph G, weights c for edges and a positive integer k.
  - **Output**: YES iff G with weights c has a TS tour of cost at most k.

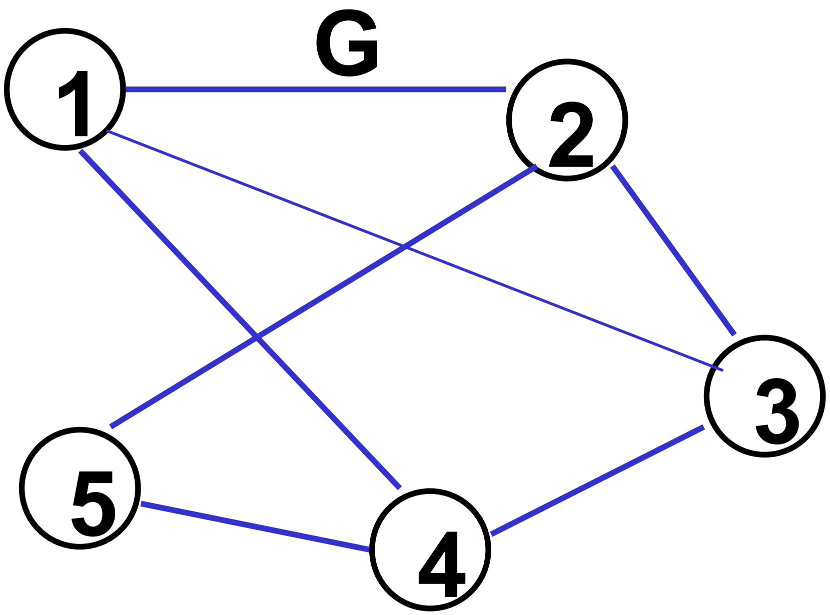
Traveling Salesman

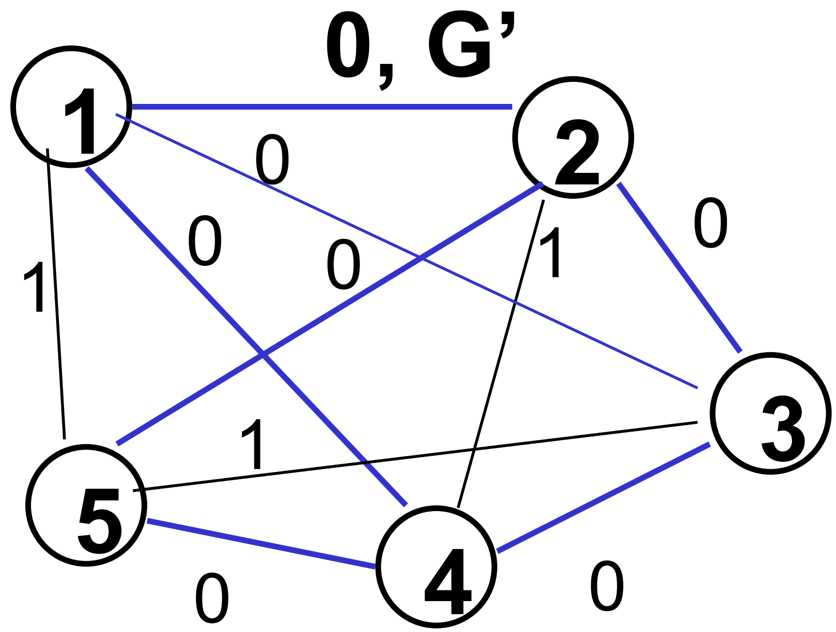
- **Theorem: TSP is NP-complete.**
- **Proof: Step 1: TSP is in NP**
  - The certificate is a representation of the tour, for example a permutation of the cities.
  - This certificate can be verified easily by checking that all cities are included exactly once and that the sum of the distances between all pairs of consecutive tour nodes is k or less.
  - This can be done in polynomial time, so TSP $\in$ NP.

The reduction

- Step 2: Select HAM-CYCLE (We will show that HAM-CYCLE $\propto$ TSP).
- Step 3: The reduction
  - Given an instance G of HAM-CYCLE, we construct a graph G' = (V, E'). G' is a complete graph and c(i,j) = 0 if (i,j) is an edge and 1 otherwise.
  - The instance of TSP is then (G', c, 0) where 0 is the bound on the cost of the tour. This conversion can be done in polynomial time (step 5).

The reduction (example)



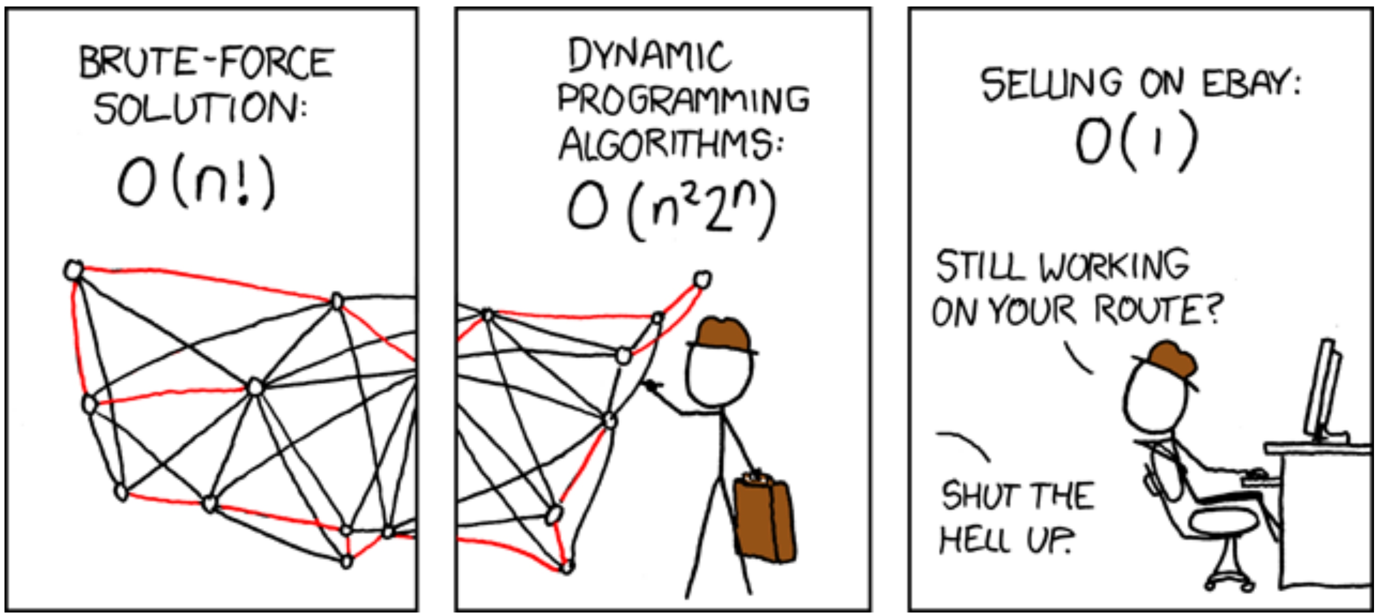


The reduction (step 4)

- **If G has a Hamiltonian cycle h, each edge in h belongs to E and thus has no cost in G'. Thus h is a tour with cost 0.**

- **If G' has a tour of cost 0, the tour must have edges from E (since any edge not in E adds 1 to the cost). Thus, the tour must be a Hamiltonian cycle in G.**

Questions?



BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.