6.045: Automata, Computability, and Complexity (GITCS)

Class 15

Nancy Lynch

Today: More Complexity Theory

- Polynomial-time reducibility, NP-completeness, and the Satisfiability (SAT) problem
- Topics:
  - Introduction (Review and preview)
  - Polynomial-time reducibility, $\leq_p$
  - Clique $\leq_p$ VertexCover and vice versa
  - NP-completeness
  - SAT is NP-complete
- Reading:
  - Sipser Sections 7.4-7.5
- Next:
  - Sipser Sections 7.4-7.5

Introduction

Introduction

- P = { L | there is some polynomial-time deterministic Turing machine that decides L }
- NP = { L | there is some polynomial-time nondeterministic Turing machine that decides L }
- Alternatively, $L \in NP$ if and only if ($\exists$ V, a polynomial-time verifier) ($\exists$ p, a polynomial) such that:

  $x \in L$ iff $(\exists\, c, |c| \le p(|x|))$ [V(x, c) accepts]

  certificate

- To show that $L \in NP$, we need only exhibit a suitable verifier V and show that it works (which requires saying what the certificates are).
- $P \subseteq NP$, but it's not known whether P = NP.

Introduction

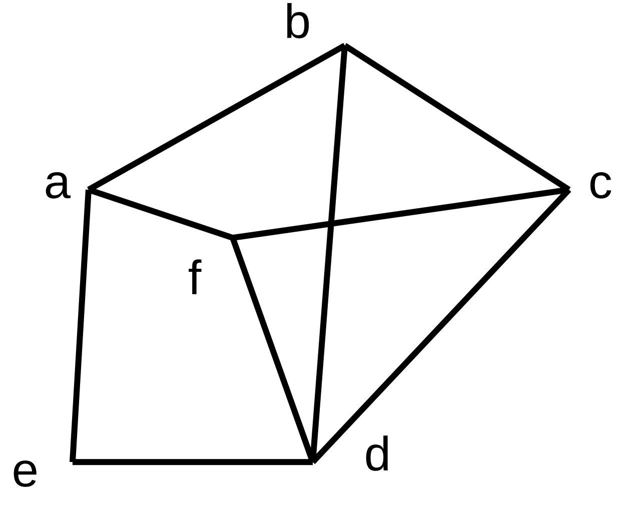
- P = { L | $\exists$ poly-time deterministic TM that decides L }
- NP = { L | $\exists$ poly-time nondeterministic TM that decides L }
- L $\in$ NP if and only if ($\exists$ V, poly-time verifier) ($\exists$ p, poly)

  x $\in$ L iff ($\exists$ c, $|c| \leq p(|x|)$) [V(x, c) accepts]

- Some languages are in NP, but are not known to be in P (and are not known to not be in P):
  - SAT = { $< \phi >$ | ϕ is a satisfiable Boolean formula }
  - 3COLOR = { $< G >$ | G is an (undirected) graph whose vertices can be colored with $\leq$ 3 colors with no 2 adjacent vertices colored the same }
  - CLIQUE = { $< G, k >$ | G is a graph with a k-clique }
  - VERTEX-COVER = { $< G, k >$ | G is a graph having a vertex cover of size k }

CLIQUE

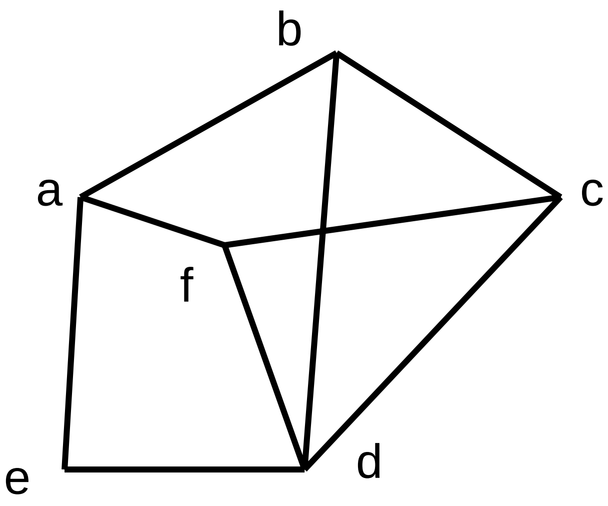
- CLIQUE = { < G, k > | G is a graph with a k-clique }
- k-clique: k vertices with edges between all pairs in the clique.
- In NP, not known to be in P, not known to not be in P.



- 3-cliques: { b, c, d }, { c, d, f }
- Cliques are easy to verify, but may be hard to find.

CLIQUE

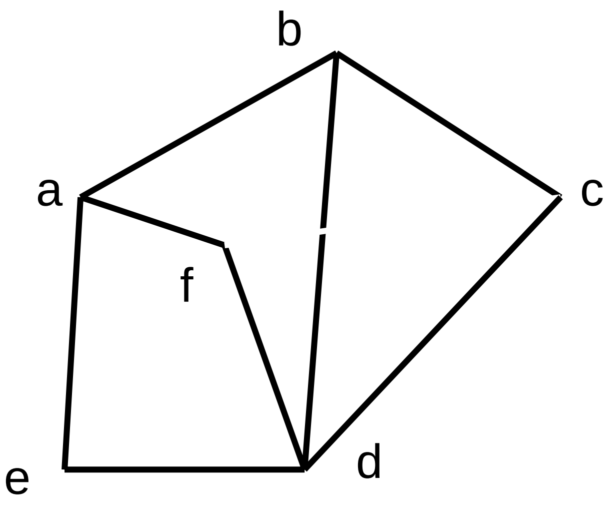
- CLIQUE = { < G, k > | G is a graph with a k-clique }



- Input to the VC problem: < G, 3 >
- Certificate, to show that < G, 3 > $\in$ CLIQUE, is { b, c, d } (or { c, d, f }).
- Polynomial-time verifier can check that { b, c, d } is a 3-clique.

VERTEX-COVER

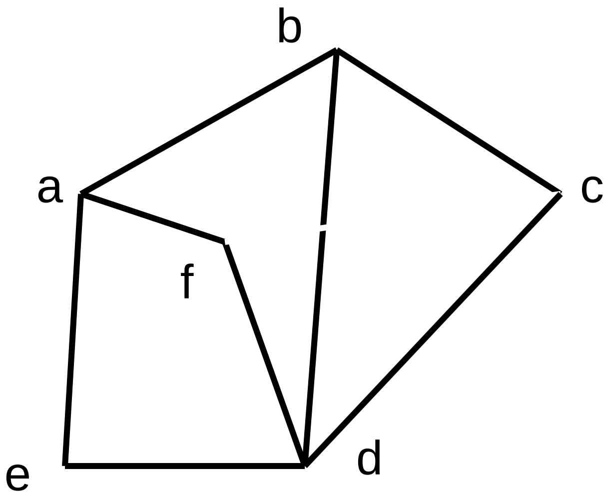
- VERTEX-COVER = { < G, k > | G is a graph with a vertex cover of size k }
- Vertex cover of G = (V, E): A subset C of V such that, for every edge (u,v) in E, either u $\in$ C or v $\in$ C.
  - A set of vertices that "covers" all the edges.
- In NP, not known to be in P, not known to not be in P.



- 3-vc: { a, b, d }
- Vertex covers are easy to verify, may be hard to find.

VERTEX-COVER

- VERTEX-COVER = { < G, k > | G is a graph with a vertex cover of size k }



- Input to the VC problem: < G, 3 >
- Certificate, to show that < G, 3 > $\in$ VC, is { a, b, d }.
- Polynomial-time verifier can check that { a, b, d } is a 3-vertex-cover.

Introduction

- Languages in NP, not known to be in P, not known to not be in P:
  - SAT = { < ϕ > | ϕ is a satisfiable Boolean formula }
  - 3COLOR = { < G > | G is a graph whose vertices can be colored with $\leq$ 3 colors with no 2 adjacent vertices colored the same }
  - CLIQUE = { < G, k > | G is a graph with a k-clique }
  - VERTEX-COVER = { < G, k > | G is a graph with a vc of size k }

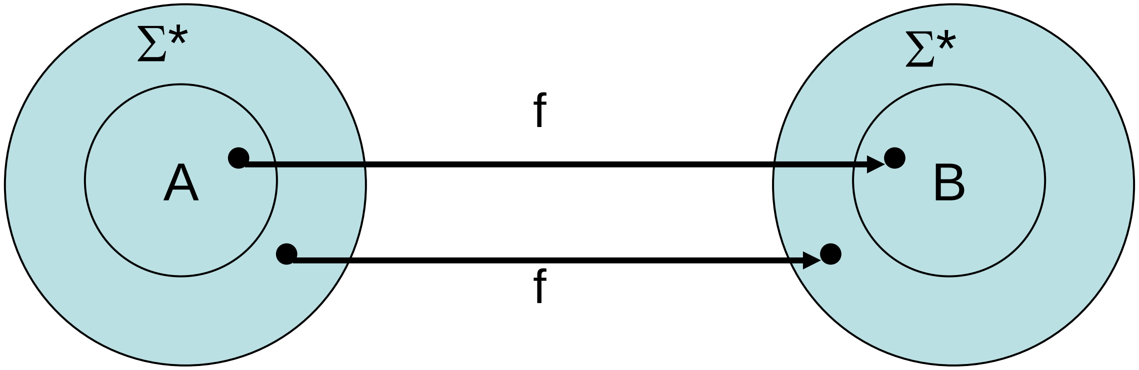
- There are many problems like these, where some structure seems hard to find, but is easy to verify.
- Q: Are these easy (in P) or hard (not in P)?
- Not yet known. We don't yet have the math tools to answer this question.
- We can say something useful to reduce the apparent diversity of such problems---that many such problems are "reducible" to each other.
- So in a sense, they are the "same problem".

Polynomial-Time Reducibility

Polynomial-Time Reducibility

- Definition: $A \subseteq \Sigma^*$ is polynomial-time reducible to $B \subseteq \Sigma^*$, $A \leq_p B$, provided there is a polynomial-time computable function $f: \Sigma^* \rightarrow \Sigma^*$ such that:

  $(\forall w)$ [$w \in A$ if and only if $f(x) \in B$]



- Extends to different alphabets Σ_1 and Σ_2.
- Same as mapping reducibility, $\leq_m$, but with a polynomial-time restriction.

Polynomial-Time Reducibility

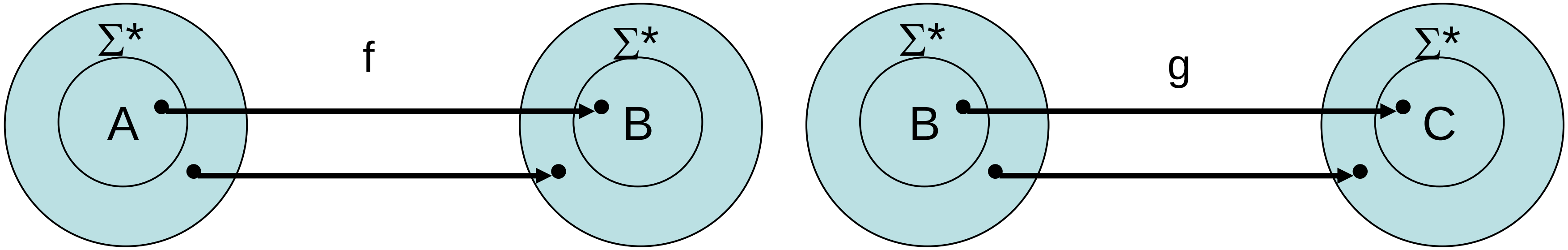
- Definition: $A \subseteq \Sigma^*$ is polynomial-time reducible to $B \subseteq \Sigma^*$, $A \leq_p B$, provided there is a polynomial-time computable function $f: \Sigma^* \rightarrow \Sigma^*$ such that:

$$(\forall w)\ [\ w \in A \text{ if and only if } f(x) \in B\ ]$$

- Theorem: (Transitivity of $\leq_p$)

  If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.

- Proof:
  - Let f be a polynomial-time reducibility function from A to B.
  - Let g be a polynomial-time reducibility function from B to C.

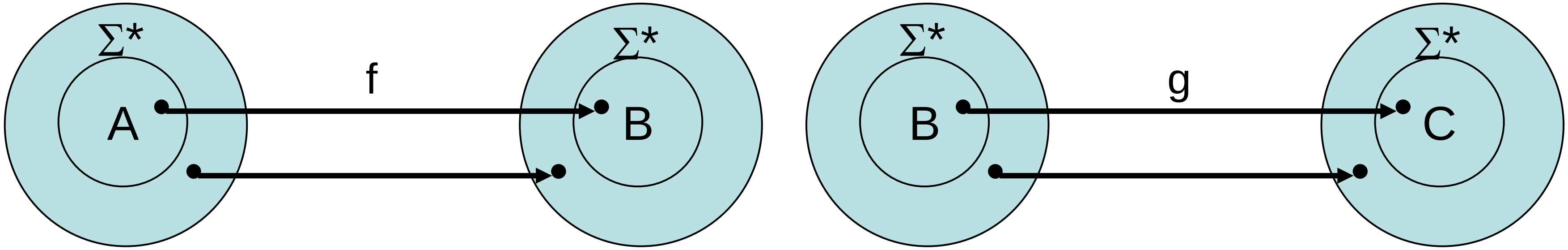


Polynomial-Time Reducibility

- Definition: $A \le_p B$, provided there is a polynomial-time computable function $f: \Sigma^* \rightarrow \Sigma^*$ such that:

$$(\forall w)\ [\ w \in A \text{ if and only if } f(w) \in B\ ]$$

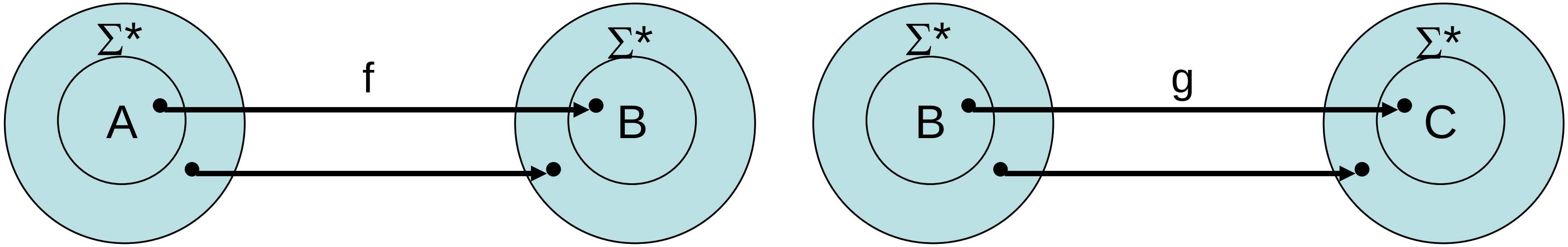
- Theorem: If $A \le_p B$ and $B \le_p C$ then $A \le_p C$.
- Proof:
  - Let f be a polynomial-time reducibility function from A to B.
  - Let g be a polynomial-time reducibility function from B to C.



  - Define $h(w) = g(f(w))$.
  - Then $w \in A$ if and only if $f(w) \in B$ if and only if $\underbrace{g(f(w))}_{\boxed{h(w)}} \in C$.
  - h is poly-time computable:

Polynomial-Time Reducibility

- Theorem: If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.
- Proof:
  - Let f be a polynomial-time reducibility function from A to B.
  - Let g be a polynomial-time reducibility function from B to C.



  - Define h(w) = g(f(w)).
  - h is poly-time computable:
    - |f(w)| is bounded by a polynomial in |w|.
    - Time to compute g(f(w)) is bounded by a polynomial in |f(w)|, and therefore by a polynomial in |w|.
    - Uses the fact that substituting one polynomial for the variable in another yields yet another polynomial.

Polynomial-Time Reducibility

- Definition: $A \leq_p B$, provided there is a polynomial-time computable function f: $\Sigma^* \rightarrow \Sigma^*$ such that:

  $$(\forall w)\ [\ w \in A \text{ if and only if } f(x) \in B\ ]$$

- Theorem: If $A \leq_p B$ and $B \in P$ then $A \in P$.
- Proof:
  - Let f be a polynomial-time reducibility function from A to B.
  - Let M be a polynomial-time decider for B.
  - To decide whether $w \in A$:
    - Compute x = f(w).
    - Run M to decide whether $x \in B$, and accept / reject accordingly.
  - Polynomial time.

- Corollary: If $A \leq_p B$ and A is not in P then B is not in P.
- Easiness propagates downward, hardness propagates upward.

Polynomial-Time Reducibility

- Can use $\leq_p$ to relate the difficulty of two problems:
- Theorem: If $A \leq_p B$ and $B \leq_p A$ then either both A and B are in P or neither is.
- Also, for problems in NP:
- Theorem: If $A \leq_p B$ and $B \in NP$ then $A \in NP$.
- Proof:
  - Let f be a polynomial-time reducibility function from A to B.
  - Let M be a polynomial-time nondeterministic TM that decides B.
    - Poly-bounded on all branches.
    - Accepts on at least one branch iff and only if input string is in B.
  - NTM M′ to decide membership in A:
  - On input w:
    - Compute $x = f(w)$; $|x|$ is bounded by a polynomial in $|w|$.
    - Run M on x and accept/reject (on each branch) if M does.
  - Polynomial time-bounded NTM.

Polynomial-Time Reducibility

- Theorem: If $A \leq_p B$ and $B \in NP$ then $A \in NP$.
- Proof:
  - Let f be a polynomial-time reducibility function from A to B.
  - Let M be a polynomial-time nondeterministic TM that decides B.
  - NTM M′ to decide membership in A:
  - On input w:
    - Compute $x = f(w)$; $|x|$ is bounded by a polynomial in $|w|$.
    - Run M on x and accept/reject (on each branch) if M does.
  - Polynomial time-bounded NTM.
  - Decides membership in A:
    - M′ has an accepting branch on input w

      iff M has an accepting branch on f(w), by definition of M′,

      iff $f(w) \in B$, since M decides B,

      iff $w \in A$, since $A \leq_p B$ using f.
  - So M′ is a poly-time NTM that decides A, $A \in NP$.

Polynomial-Time Reducibility

- Theorem: If $A \leq_p B$ and $B \in NP$ then $A \in NP$.
- Corollary: If $A \leq_p B$ and A is not in NP, then B is not in NP.

Polynomial-Time Reducibility

- A technical result (curiosity):
- Theorem: If $A \in P$ and B is any nontrivial language (meaning not $\emptyset$, not Σ^*), then $A \leq_p B$.
- Proof:
  - Suppose $A \in P$.
  - Suppose B is a nontrivial language; pick $b_0 \in B$, $b_1 \in B^c$.
  - Define $f(w) = b_0$ if $w \in A$, b_1 if w is not in A.
  - f is polynomial-time computable; why?
  - Because A is polynomial time decidable.
  - Clearly $w \in A$ if and only if $f(w) \in B$.
  - So $A \leq_p B$.
- Trivial reduction: All the work is done by the decider for A, not by the reducibility and the decider for B.

CLIQUE and VERTEX-COVER

CLIQUE and VERTEX-COVER

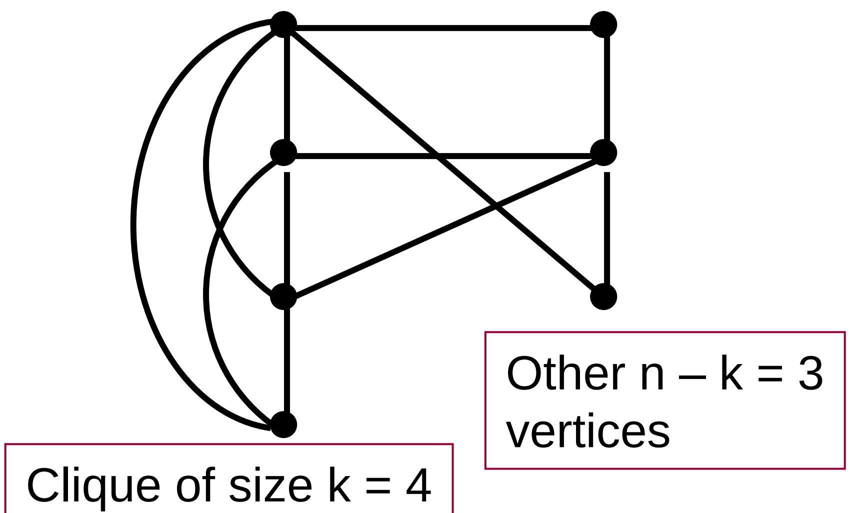
- Two illustrations of $\leq_p$.
- Both CLIQUE and VC are in NP, not known to be in P, not known to not be in P.
- However, we can show that they are essentially equivalent: polynomial-time reducible to each other.
- So, although we don't know how hard they are, we know they are (approximately) equally hard.
  - E.g., if either is in P, then so is the other.
- Theorem: CLIQUE $\leq_p$ VC.
- Theorem: VC $\leq_p$ CLIQUE.

CLIQUE and VERTEX-COVER

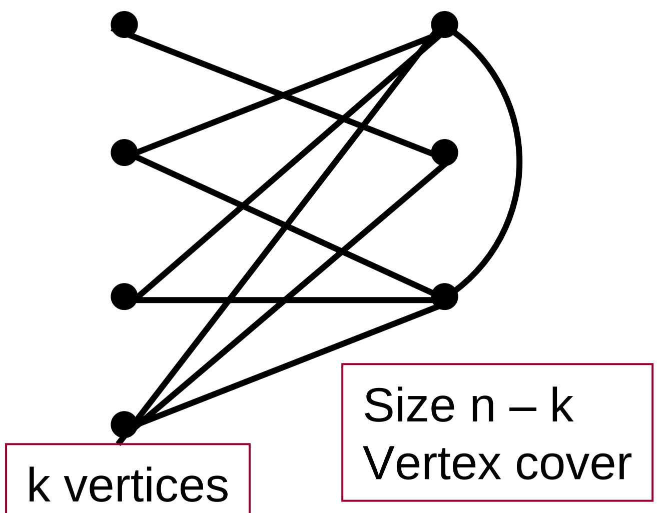
- Theorem: CLIQUE $\leq_p$ VC.
- Proof:
  - Given input < G, k > for CLIQUE, transform to input < G′, k′ > for VC, in poly time, so that:

    < G, k > $\in$ CLIQUE if and only if < G′, k′ > $\in$ VC.
- Example:

G = (V, E), k = 4

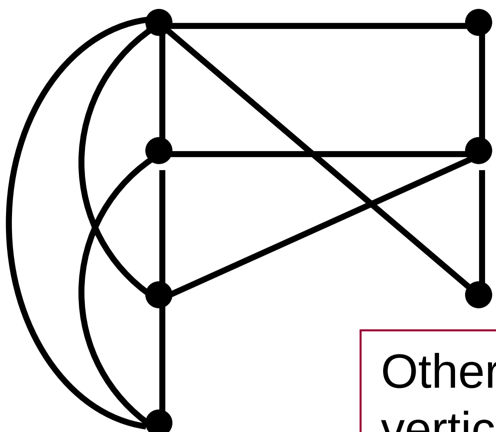


G′ = (V, E′), k′ = n – k = 3



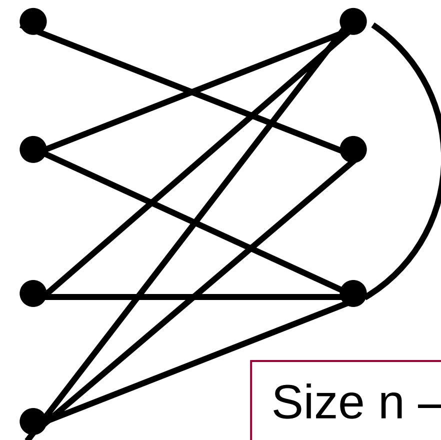
CLIQUE and VERTEX-COVER

- < G, k > $\in$ CLIQUE if and only if < G′, k′ > $\in$ VC.
- Example: G = (V, E), k = 4, G′ = (V, E′), k′ = n – k = 3



Clique of size k = 4

Other n – k = 3 vertices



k vertices

Size n – k Vertex cover

- E′ = (V $\times$ V) – E, complement of edge set
- G has clique of size 4 (left nodes), G′ has a vertex cover of size 7 – 4 = 3 (right nodes).
- All edges between 2 nodes on left are in E, hence not in E′, so right nodes cover all edges in E′.

CLIQUE and VERTEX-COVER

- Theorem: CLIQUE $\leq_p$ VC.
- Proof:
  - Given input < G, k > for CLIQUE, transform to input < G′, k′ > for VC, in poly time, so that < G, k > $\in$ CLIQUE iff < G′, k′ > $\in$ VC.
  - General transformation: f(< G, k >), where G = (V, E) and |V| = n,

    = < G′, n-k >, where G′ = (V, E′) and E′ = (V $\times$ V) – E.
  - Transformation is obviously polynomial-time.
  - Claim: G has a k-clique iff G′ has a size (n-k) vertex cover.
  - Proof of claim: Two directions:

    $\Rightarrow$ Suppose G has a k-clique, show G′ has an (n-k)-vc.
    - Suppose C is a k-clique in G.
    - V – C is an (n-k)-vc in G′:
      - Size is obviously right.
      - All edges between nodes in C appear in G, so all are missing in G′.
      - So nodes in V-C cover all edges of G′.

CLIQUE and VERTEX-COVER

- Theorem: CLIQUE $\leq_p$ VC.
- Proof:
  - Given input < G, k > for CLIQUE, transform to input < G′, k′ > for VC, in poly time, so that < G, k > $\in$ CLIQUE iff < G′, k′ > $\in$ VC.
  - General transformation: f(< G, k >), where G = (V, E) and |V| = n,

    = < G′, n-k >, where G′ = (V, E′) and E′ = (V $\times$ V) – E.
  - Claim: G has a k-clique iff G′ has a size (n-k) vertex cover.
  - Proof of claim: Two directions:

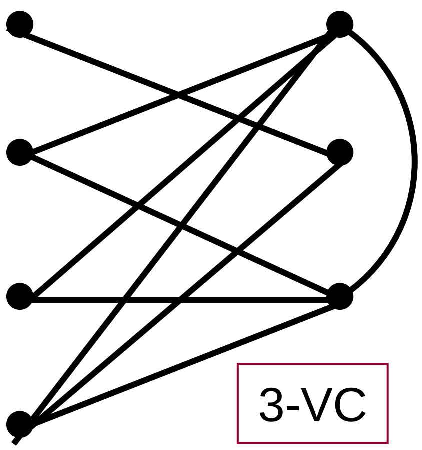
    $\Leftarrow$ Suppose G′ has an (n-k)-vc, show G has a k-clique.
    - Suppose D is an (n-k)-vc in G′.
    - V – D is a k-clique in G:
      - Size is obviously right.
      - All edges between nodes in V-D are missing in G′, so must appear in G.
      - So V-D is a clique in G.

CLIQUE and VERTEX-COVER

- Theorem: VC $\leq_p$ CLIQUE.
- Proof: Almost the same.
  - Given input < G, k > for VC, transform to input < G′, k′ > for CLIQUE, in poly time, so that:

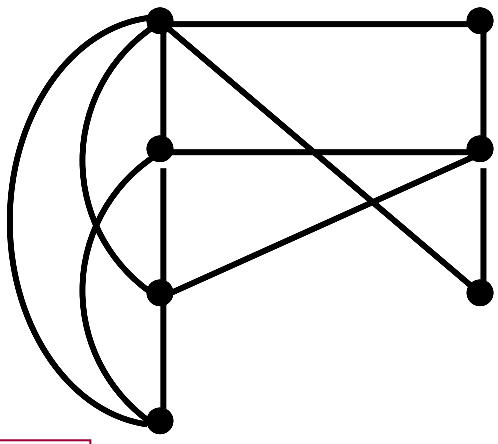
    < G, k > $\in$ VC if and only if < G′, k′ > $\in$ CLIQUE.
- Example:

G = (V, E), k = 3



3-VC

G′ = (V, E′), k′ = 4

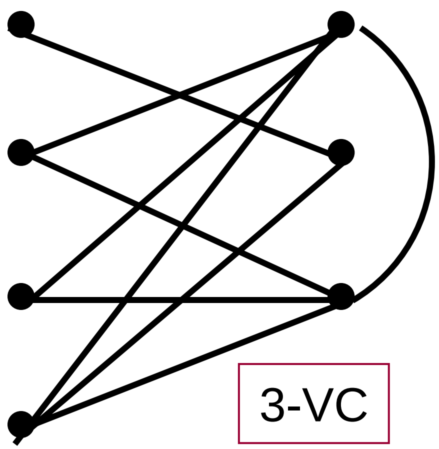


4-clique

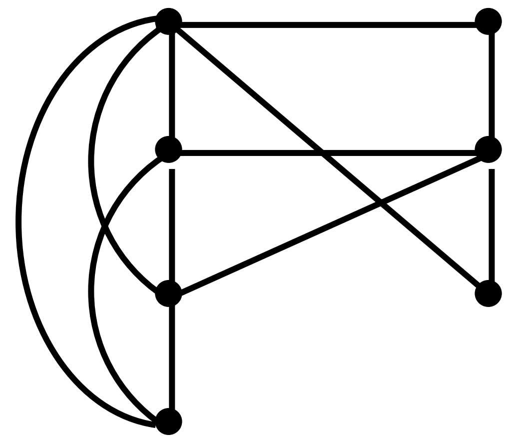
CLIQUE and VERTEX-COVER

$< G, k > \in$ VC if and only if $< G', k' > \in$ CLIQUE.

- Example: $G = (V, E)$, $k = 3$, $G' = (V, E')$, $k' = 4$



3-VC



4-clique

- $E' = (V \times V) - E$, complement of edge set
- G has a 3-vc (right nodes), G′ has clique of size 7 – 3 = 4 (left nodes).
- All edges between 2 nodes on left are missing from G, so are in G′, so left nodes form a clique in G′.

CLIQUE and VERTEX-COVER

- Theorem: VC $\leq_p$ CLIQUE.
- Proof:
  - Given input < G, k > for VC, transform to input < G′, k′ > for CLIQUE, in poly time, so that < G, k > $\in$ VC iff < G′, k′ > $\in$ CLIQUE.
  - General transformation: Same as before.

    f(< G, k >), where G = (V, E) and |V| = n,

    = < G′, n-k >, where G′ = (V, E′) and E′ = $(V \times V) - E$.
  - Claim: G has a k-vc iff G′ has an (n-k)-clique.
  - Proof of claim: Similar to before, LTTR.

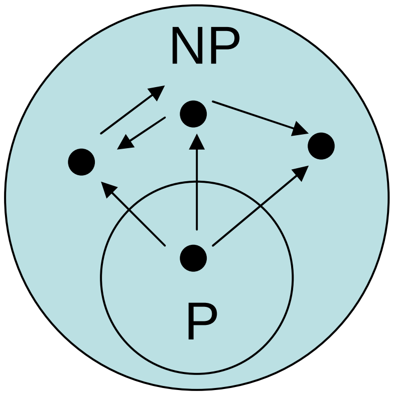
CLIQUE and VERTEX-COVER

- We have shown:
- Theorem: CLIQUE $\leq_p$ VC.
- Theorem: VC $\leq_p$ CLIQUE.
- So, they are essentially equivalent.
- Either both CLIQUE and VC are in P or neither is.

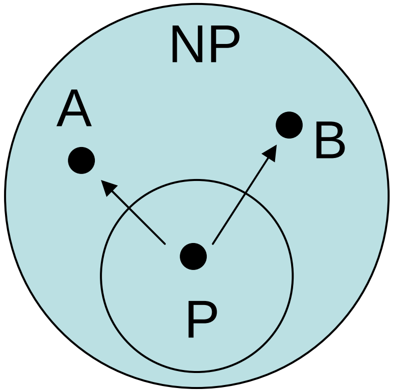
NP-Completeness

NP-Completeness

- $\leq_p$ allows us to relate problems in NP, saying which allow us to solve which others efficiently.
- Even though we don't know whether all of these problems are in P, we can use $\leq_p$ to impose some structure on the class NP:

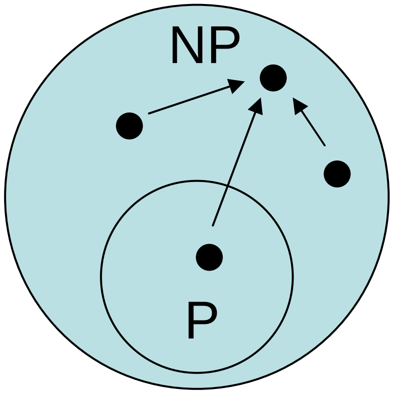


- $A \rightarrow B$ here means $A \leq_p B$.
- Sets in NP – P might not be totally ordered by $\leq_p$: we might have A, B with neither $A \leq_p B$ nor $B \leq_p A$:



NP-Completeness

- Some languages in NP are hardest, in the sense that every language in NP is $\leq_p$-reducible to them.
- Call these NP-complete.
- Definition: Language B is NP-complete if both of the following hold:
  - (a) $B \in NP$, and
  - (b) For any language $A \in NP$, $A \leq_p B$.



- Sometimes, we consider languages that aren't, or might not be, in NP, but to which all NP languages are reducible.
- Call these NP-hard.
- Definition: Language B is NP-hard if, for any language $A \in NP$, $A \leq_p B$.

NP-Completeness

- Today, and next time, we'll:
  - Give examples of interesting problems that are NP-complete, and
  - Develop methods for showing NP-completeness.

- Theorem: $\exists$B, B is NP-complete.
  - There is at least one NP-complete problem.
  - We'll show this later.
- Theorem: If A, B, are NP-complete, then A $\leq_p$ B.
  - Two NP-complete problems are essentially equivalent (up to $\leq_p$).
- Proof: A $\in$ NP, B is NP-hard, so A $\leq_p$ B by definition.

NP-Completeness

- Theorem: If some NP-complete language is in P, then P = NP.
  - That is, if a polynomial-time algorithm exists for any NP-complete problem, then the entire class NP collapses into P.
  - Polynomial algorithms immediately arise for all problems in NP.
- Proof:
  - Suppose B is NP-complete and B $\in$ P.
  - Let A be any language in NP; show A $\in$ P.
  - We know A $\leq_p$ B since B is NP-complete.
  - Then A $\in$ P, since B $\in$ P and "easiness propagates downward".
  - Since every A in NP is also in P, NP $\subseteq$ P.
  - Since P $\subseteq$ NP, it follows that P = NP.

NP-Completeness

- Theorem: The following are equivalent.
  1. P = NP.
  2. Every NP-complete language is in P.
  3. Some NP-complete language is in P.
- Proof:

  $1 \Rightarrow 2$:
  - Assume P = NP, and suppose that B is NP-complete.
  - Then $B \in NP$, so $B \in P$, as needed.

  $2 \Rightarrow 3$:
  - Immediate because there is at least NP-complete language.

  $3 \Rightarrow 1$:
  - By the previous theorem.

Beliefs about P vs. NP

- Most theoretical computer scientists believe P $\neq$ NP.
- Why?
- Many interesting NP-complete problems have been discovered over the years, and many smart people have tried to find fast algorithms; no one has succeeded.
- The problems have arisen in many different settings, including logic, graph theory, number theory, operations research, games and puzzles.
- Entire book devoted to them [Garey, Johnson].
- All these problems are essentially the same since all NP-complete problems are polynomial-reducible to each other.
- So essentially the same problem has been studied in many different contexts, by different groups of people, with different backgrounds, using different methods.

Beliefs about P vs. NP

- Most theoretical computer scientists believe $P \neq NP$.
- Because many smart people have tried to find fast algorithms and no one has succeeded.
- That doesn't mean $P \neq NP$; this is just some kind of empirical evidence.
- The essence of why NP-complete problems seem hard:
  - They have NP structure:

    $$x \in L \text{ iff } (\exists\, c, |c| \leq p(|x|))\ [\ V(\ x, c\ ) \text{ accepts }],$$

    where V is poly-time.
  - Guess and verify.
  - Seems to involve exploring a tree of possible choices, exponential blowup.
- However, no one has yet succeeded in proving that they actually are hard!
  - We don't have sharp enough methods.
  - So in the meantime, we just show problems are NP-complete.

Satisfiability is NP-Complete

Satisfiability is NP-Complete

- SAT = { < ϕ > | ϕ is a satisfiable Boolean formula }
- Definition: (Boolean formula):
  - Variables: $x, x_1, x_2, \ldots, y, \ldots, z, \ldots$
    - Can take on values 1 (true) or 0 (false).
  - Literal: A variable or its negated version: $x, \neg x, \neg x_1, \ldots$
  - Operations: $\wedge \ \vee \ \neg$
  - Boolean formula: Constructed from literals using operations, e.g.:

    $\phi = x \wedge ((y \wedge z) \vee (\neg y \wedge \neg z)) \wedge \neg(x \wedge z)$
- Definition: (Satisfiability):
  - A Boolean formula is satisfiable iff there is an assignment of 0s and 1s to the variables that makes the entire formula evaluate to 1 (true).

Satisfiability is NP-Complete

- SAT = { < ϕ > | ϕ is a satisfiable Boolean formula }
- Boolean formula: Constructed from literals using operations, e.g.:

  $$\phi = x \wedge ((y \wedge z) \vee (\neg y \wedge \neg z)) \wedge \neg(x \wedge z)$$

- A Boolean formula is satisfiable iff there is an assignment of 0s and 1s to the variables that makes the entire formula evaluate to 1 (true).
- Example: ϕ above
  - Satisfiable, using the assignment x = 1, y = 0, z = 0.
  - So $\phi \in$ SAT.
- Example: $x \wedge ((y \wedge z) \vee (\neg y \wedge z)) \wedge \neg(x \wedge z)$
  - Not in SAT.
  - x must be set to 1, so z must = 0.

Satisfiability is NP-Complete

- SAT = { < ϕ > | ϕ is a satisfiable Boolean formula }
- Theorem: SAT is NP-complete.
- Lemma 1: SAT $\in$ NP.
- Lemma 2: SAT is NP-hard.
- Proof of Lemma 1:
  - Recall: L $\in$ NP if and only if ($\exists$ V, poly-time verifier) ($\exists$ p, poly)

    x $\in$ L iff ($\exists$ c, |c| $\leq$ p(|x|)) [V(x, c) accepts]
  - So, to show SAT $\in$ NP, it's enough to show ($\exists$ V) ($\exists$ p)

    ϕ $\in$ SAT iff ($\exists$ c, |c| $\leq$ p(|x|)) [V(ϕ, c) accepts]
  - We know: ϕ $\in$ SAT iff there is an assignment to the variables such that ϕ with this assignment evaluates to 1.
  - So, let certificate c be the assignment.
  - Let verifier V take a formula ϕ and an assignment c and accept exactly if ϕ with c evaluates to true.
  - Evaluate ϕ bottom-up, takes poly time.

Satisfiability is NP-Complete

- Lemma 2: SAT is NP-hard.
- Proof of Lemma 2:
  - Need to show that, for any $A \in NP$, $A \leq_p SAT$.
  - Fix $A \in NP$.
  - Construct a poly-time f such that

$$w \in A \text{ if and only if } f(w) \in SAT.$$

A formula, write it as ϕ_w.

  - By definition, since $A \in NP$, there is a nondeterministic TM M that decides A in polynomial time.
  - Fix polynomial p such that M on input w always halts, on all branches, in time $\leq p(|w|)$; assume $p(|w|) \geq |w|$.
  - $w \in A$ if and only if there is an accepting computation history (CH) of M on w.

Satisfiability is NP-Complete

- Lemma 2: SAT is NP-hard.
- Proof, cont'd:
  - Need $w \in A$ if and only if $f(w)$ ($= \phi_w$) $\in$ SAT.
  - $w \in A$ if and only if there is an accepting CH of M on w.
  - So we must construct formula ϕ_w to be satisfiable iff there is an accepting CH of M on w.
  - Recall definitions of computation history and accepting computation history from Post Correspondence Problem: $\#\ C_0\ \#\ C_1\ \#\ C_2 \ldots$
    - Configurations include tape contents, state, head position.
  - We construct ϕ_w to describe an accepting CH.
  - Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ as usual.
  - Instead of lining up configs in a row as before, arrange in $(p(|w|) + 1)$ row $\times$ $(p(|w|) + 3)$ column matrix:

Proof that SAT is NP-hard

- ϕ_w will be satisfiable iff there is an accepting CH of M on w.
- Let M = (Q, Σ, Γ, δ, q_0, q_{acc}, q_{rej}).
- Arrange configs in (p(|w|) + 1) $\times$ (p(|w|) + 3) matrix:

$$
\begin{array}{cccccccccccc}
\# & q_0 & w_1 & w_2 & w_3 & \ldots & w_n & -- & -- & \ldots & -- & \# \\
\# & \ldots & & & & & & & & & & \# \\
\# & \ldots & & & & & & & & & & \# \\
\vdots & & & & & & & & & & & \vdots \\
\# & \ldots & & & & & & & & & & \#
\end{array}
$$

- Successive configs, ending with accepting config.
- Assume WLOG that each computation takes exactly p(|w|) steps, so we use p(|w|) + 1 rows.
- p(|w|) + 3 columns: p(|w|) for the interesting portion of the tape, one for head and state, two for endmarkers.

Proof that SAT is NP-hard

- ϕ_w is satisfiable iff there is an accepting CH of M on w.
- Entries in the matrix are represented by Boolean variables:
  - Define $C = Q \cup \Gamma \cup \{ \# \}$, alphabet of possible matrix entries.
  - Variable $x_{i,j,c}$ represents "the entry in position (i, j) is c".
- Define ϕ_w as a formula over these $x_{i,j,c}$ variables, satisfiable if and only if there is an accepting computation history for w (in matrix form).
- Moreover, an assignment of values to the $x_{i,j,c}$ variables that satisfies ϕ_w will correspond to an encoding of an accepting computation.
- Specifically, $\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$, where:
  - ϕ_{cell} : There is exactly one value in each matrix location.
  - ϕ_{start} : The first row represents the starting configuration.
  - ϕ_{accept} : The last row is an accepting configuration.
  - ϕ_{move} : Successive rows represent allowable moves of M.

ϕ_{cell}

- For each position (i,j), write the conjunction of two formulas:

  $\bigvee_{c \in C} x_{i,j,c}$: Some value appears in position (i,j).

  $\bigwedge_{c,\, d \in C,\, c \neq d} (\neg x_{i,j,c} \vee \neg x_{i,j,d})$: Position (i,j) doesn't contain two values.
- ϕ_{cell}: Conjoin formulas for all positions (i,j).

- Easy to construct the entire formula ϕ_{cell} given w input.
- Construct it in polynomial time.
- Sanity check: Length of formula is polynomial in |w|:
  - O($(p(|w|)^2$) subformulas, one for each (i,j).
  - Length of each subformula depends on C, O($|C|^2$).

ϕ_{start}

- The right symbols appear in the first row:

$$\# \quad q_0 \quad w_1 \quad w_2 \quad w_3 \quad \ldots \quad w_n \quad \text{--} \quad \text{--} \quad \ldots \quad \text{--} \quad \#$$

$$\phi_{start}: \ x_{1,1,\#} \wedge x_{1,2,q0} \wedge x_{1,3,w1} \wedge x_{1,4,w2} \wedge \cdots$$
$$\wedge\ x_{1,n+2,wn} \wedge x_{1,n+3,\text{--}} \wedge \cdots$$
$$\wedge\ x_{1,p(n)+2,\text{--}} \wedge x_{1,p(n)+3,\#}$$

ϕ_{accept}

- For each j, $2 \leq j \leq p(|w|) + 2$, write the formula:

$$x_{p(|w|)+1,j,qacc}$$

- q_{acc} appears in position j of the last row.
- ϕ_{accept}: Take disjunction (or) of all formulas for all j.
- That is, q_{acc} appears in some position of the last row.

ϕ_{move}

- As for PCP, correct moves depend on correct changes to local portions of configurations.
- It's enough to consider 2×3 rectangles:
- If every 2×3 rectangle is "good", i.e., consistent with the transitions, then the entire matrix represents an accepting CH.
- For each position (i,j), $1 \leq i \leq p(|w|)$, $1 \leq j \leq p(|w|)+1$, write a formula saying that the rectangle with upper left at (i,j) is "good".
- Then conjoin all of these, $O(p(|w|)^2)$ clauses.
- Good tiles for (i,j), for a, b, c in Γ:

a	b	c
a	b	c

#	a	b
#	a	b

a	b	#
a	b	#

ϕ_{move}

- Other good tiles are defined in terms of the nondeterministic transition function δ.
- E.g., if $\delta(q_1, a)$ includes tuple (q_2, b, L), then the following are good:
  - Represents the move directly; for any c:
  - Head moves left out of the rectangle; for any c, d:
  - Head is just to the left of the rectangle; for any c, d:
  - Head at right; for any c, d, e:
  - And more, for #, etc.
- Analogously if $\delta(q_1, a)$ includes (q_2, b, R).
- Since M is nondeterministic, $\delta(q_1, a)$ may contain several moves, so include all the tiles.

c	q_1	a
q_2	c	b

q_1	a	c
d	b	c

a	c	d
b	c	d

d	c	q_1
d	q_2	c

e	d	c
e	d	q_2

ϕ_{move}

- The good tiles give partial constraints on the computation.
- When taken together, they give enough constraints so that only a correct CH can satisfy them all.
- The part (conjunct) of ϕ_{move} for (i,j) should say that the rectangle with upper left at (i,j) is good:
- It is simply the disjunction (or), over all allowable tiles, of the subformula:

a1	a2	a3
b1	b2	b3

$$x_{i,j,a1} \wedge x_{i,j+1,a2} \wedge x_{i,j+2,a3} \wedge x_{i+1,j,b1} \wedge x_{i+1,j+1,b2} \wedge x_{i+1,j+2,b3}$$

- Thus, ϕ_{move} is the conjunction over all (i,j), of the disjunction over all good tiles, of the formula just above.

ϕ_{move}

- ϕ_{move} is the conjunction over all (i,j), of the disjunction over all good tiles, of the given six-term conjunctive formula.
- Q: How big is the formula ϕ_{move}?
- $O(p(|w|)^2)$ clauses, one for each (i,j) pair.
- Each clause is only constant length, O(1).
  - Because machine M yields only a constant number of good tiles.
  - And there are only 6 terms for each tile.
- Thus, length of ϕ_{move} is polynomial in |w|.
- $\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$, length also poly in |w|.

ϕ_{move}

- $\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$, length poly in |w|.
- More importantly, can produce ϕ_w from w in time that is polynomial in |w|.
- w $\in$ A if and only if M has an accepting CH for w if and only if ϕ_w is satisfiable.
- Thus, A $\leq_p$ SAT.
- Since A was any language in NP, this proves that SAT is NP-hard.
- Since SAT is in NP and is NP-hard, SAT is NP-complete.

Next time…

- NP-completeness---more examples
- Reading:
  - Sipser Sections 7.4-7.5

6.045J / 18.400J Automata, Computability, and Complexity
Spring 2011