COMP3630 / COMP6363

week 9: **Time Complexity**

This Lecture Covers Chapter 10 of HMU: Time Complexity

slides created by: Dirk Pattinson, based on material by
Peter Hoefner and Rob van Glabbeck; with improvements by Pascal Bercher

convenor & lecturer: Pascal Bercher

The Australian National University

Semester 1, 2023

Content of this Chapter

NP-Completeness of:

- *CNFSAT*
- *3SAT*
- *CLIQUE*
- *HAMPATH* (Hamiltonion Path)
- *Node Cover*
- *Independent Set*

Additional Reading: Chapter 10 of HMU.

Cook's Theorem (*SAT* is **NP**-Complete)

- *SAT* is the granddaddy of all **NP**-complete problems.
  (That's because it's the first that was proved **NP**-complete,)

Cook's Theorem (*SAT* is **NP**-Complete)

- *SAT* is the granddaddy of all **NP**-complete problems.
  (That's because it's the first that was proved **NP**-complete,)
- Cook's theorem gives a "generic reduction" for every problem in **NP** to *SAT*.
  More formally, for each $A \in \mathbf{NP}$ we have $A \leq_{\mathbf{P}} B$.
- So SAT is at least as hard as any other problem in **NP**.
- Since SAT is also in **NP**, it's **NP**-complete.

Cook's Theorem (*SAT* is **NP**-Complete)

- *SAT* is the granddaddy of all **NP**-complete problems.
  (That's because it's the first that was proved **NP**-complete,)
- Cook's theorem gives a "generic reduction" for every problem in **NP** to *SAT*.
  More formally, for each $A \in \mathbf{NP}$ we have $A \leq_{\mathbf{P}} B$.
- So SAT is at least as hard as any other problem in **NP**.
- Since SAT is also in **NP**, it's **NP**-complete.
- Many people have worked on the *SAT* problem, and there are now very efficient (SAT) solversfor it.
- People frequently translate **NP**-complete problems to propositional logic, and then attack them with these general solvers! (Even if the problem is computationally harder, this <u>might</u> be efficient – although we suffer from a blow-up.)

Cook's Theorem (*SAT* is **NP**-Complete)

- *SAT* is the granddaddy of all **NP**-complete problems.
  (That's because it's the first that was proved **NP**-complete,)
- Cook's theorem gives a "generic reduction" for every problem in **NP** to *SAT*.
  More formally, for each $A \in \mathbf{NP}$ we have $A \leq_{\mathbf{P}} B$.
- So SAT is at least as hard as any other problem in **NP**.
- Since SAT is also in **NP**, it's **NP**-complete.
- Many people have worked on the *SAT* problem, and there are now very efficient (SAT) solversfor it.
- People frequently translate **NP**-complete problems to propositional logic, and then attack them with these general solvers! (Even if the problem is computationally harder, this might be efficient – although we suffer from a blow-up.)
- But SAT also serves as a good problem to reduce from! (We look at variants of it.)

CNFSAT

CNFSAT is a special case of *SAT*.

$$\mathit{CNFSAT} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable cnf formula } \}$$

where a Boolean formula is in cnf (for conjunctive normal form) if it is (also) generated by the grammar

$$\phi \to (c) \mid (c) \wedge \phi \qquad c \to \ell \mid \ell \vee c$$
$$\ell \to p \mid \neg p \qquad p \to x \mid y \mid \ \ldots$$

We call cs clauses, ℓs literals, and ps propositions.

Intuitively, a cnf is simply a conjunction of disjunctions (also called clauses).

CNFSAT

CNFSAT is a special case of *SAT*.

$$CNFSAT = \{\ \langle\phi\rangle \mid \phi \text{ is a satisfiable cnf formula }\}$$

where a Boolean formula is in cnf (for conjunctive normal form) if it is (also) generated by the grammar

$$\phi \to (c) \mid (c) \wedge \phi \qquad\qquad c \to \ell \mid \ell \vee c$$
$$\ell \to p \mid \neg p \qquad\qquad p \to x \mid y \mid \ldots$$

We call cs clauses, ℓs literals, and ps propositions.

Intuitively, a cnf is simply a conjunction of disjunctions (also called clauses).

Example 10.2.1

$(x \vee z) \wedge (\neg y \vee z)$ is a cnf for the Boolean formula $(x \wedge \neg y) \vee z$.

CNFSAT is **NP**-Complete

Clearly *CNFSAT* is in **NP** because we can use the same certificate for ϕ in cnf as we would for the same ϕ in *SAT*. (I.e., just guess an assignment and verify.)

CNFSAT is **NP**-Complete

Clearly *CNFSAT* is in **NP** because we can use the same certificate for ϕ in cnf as we would for the same ϕ in *SAT*. (I.e., just guess an assignment and verify.)

Giving a **P** reduction from *SAT* to *CNFSAT* is tricky.

A straight-forward translation of Boolean formulae into equivalent cnf may result in an exponential blow-up, meaning that this approach is useless.

CNFSAT is **NP**-Complete

Clearly *CNFSAT* is in **NP** because we can use the same certificate for ϕ in cnf as we would for the same ϕ in *SAT*. (I.e., just guess an assignment and verify.)

Giving a **P** reduction from *SAT* to *CNFSAT* is tricky.

A straight-forward translation of Boolean formulae into equivalent cnf may result in an exponential blow-up, meaning that this approach is useless.

Instead, we recall a reduction f won't have to preserve satisfaction:

$$\forall \pi \, (\pi \models \phi \quad \Leftrightarrow \quad \pi \models f(\phi))$$

but merely satisfiability

$$\exists \pi \, (\pi \models \phi) \quad \Leftrightarrow \quad \exists \pi \, (\pi \models f(\phi))$$

meaning that we're free to choose different πs for the two sides.

CNFSAT is **NP**-Hard

The translation from Boolean formulae to cnf proceeds in two steps which are both in **P**.

1. Translate to nnf (negation normal form). (A formula where each negation symbol appears only in front of propositions.)
   This is achieved by pushing all negation symbols down to propositions and eliminating two consecutive negations. (This is still satisfaction-preserving.)
2. Translate from nnf to cnf. (This merely preserves satisfiability.)

Pushing Down $\neg$

We use de Morgan's laws and the law of double negation to rewrite left-hand-sides to right-hand-sides:

$$\begin{aligned} \text{de Morgan on conjunctions:} \quad & \neg(\phi \land \psi) \Leftrightarrow \neg(\phi) \lor \neg(\psi) \\ \text{de Morgan on disjunctions:} \quad & \neg(\phi \lor \psi) \Leftrightarrow \neg(\phi) \land \neg(\psi) \\ \text{double-negation elimination:} \quad & \neg(\neg(\phi)) \Leftrightarrow \phi \end{aligned}$$

Example 10.2.2

$$\neg((\neg(x \lor y)) \land (\neg x \lor y)) =$$

Pushing Down ¬

We use de Morgan's laws and the law of double negation to rewrite left-hand-sides to right-hand-sides:

$$\begin{aligned} \text{de Morgan on conjunctions:} \quad & \neg(\phi \wedge \psi) \Leftrightarrow \neg(\phi) \vee \neg(\psi) \\ \text{de Morgan on disjunctions:} \quad & \neg(\phi \vee \psi) \Leftrightarrow \neg(\phi) \wedge \neg(\psi) \\ \text{double-negation elimination:} \quad & \neg(\neg(\phi)) \Leftrightarrow \phi \end{aligned}$$

Example 10.2.2

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

$$\begin{aligned} & \Leftrightarrow \neg(\neg(x \vee y)) \vee \neg(\neg x \vee y) \\ & \Leftrightarrow x \vee y \vee \neg(\neg x \vee y) \\ & \Leftrightarrow x \vee y \vee \neg(\neg x) \wedge \neg y \\ & \Leftrightarrow x \vee y \vee x \wedge \neg y \\ & \Leftrightarrow x \vee y \vee (x \wedge \neg y) \quad \text{This is a disjunction!} \end{aligned}$$

Pushing Down $\neg$ cont.

Theorem 10.2.3

Every Boolean formula ϕ is equivalent to a Boolean formula ψ in nnf. Moreover, $|\psi|$ is linear in $|\phi|$ and ψ can be constructed from ϕ in **P**.

Proof.

By induction on the number n of Boolean operators ($\wedge$, $\vee$, $\neg$) in ϕ we may show that there is an equivalent ψ in nnf with at most $2n - 1$ operators. We also have to show that the number of steps is bounded linearly and that each step has polynomial effort. $\square$

nnf $\longrightarrow$ cnf

Theorem 10.2.4

There is a constant c such that every nnf ϕ has a cnf ψ such that:

1. ψ *consists of at most* $|\phi|$ *clauses.*
2. ψ *is constructable from* ϕ *in time at most* $c|\phi|^2$.
3. $\pi \models \phi$ *iff there exists an extension* π' *of* π *satisfying* $\pi' \models \psi$, *for all interpretations* π *of the propositions in* ϕ

Thus, we can turn any nnf ψ *into cnf in polynomial time.*

Proof.

By induction on $|\phi|$. $\square$

nnf $\longrightarrow$ cnf cont.

The transformation is done by the Tseytin transformation (from 1968).
Example taken from Wikipedia.

Example 10.2.5

Let $\phi = ((p \vee q) \wedge r) \rightarrow (\neg s)$. We introduce new auxiliary variables for all subformulae:

nnf $\longrightarrow$ cnf cont.

The transformation is done by the Tseytin transformation (from 1968).
Example taken from Wikipedia.

Example 10.2.5

Let $\phi = ((p \vee q) \wedge r) \rightarrow (\neg s)$. We introduce new auxiliary variables for all subformulae:

$$x_1 \leftrightarrow \neg s \qquad x_2 \leftrightarrow p \vee q \qquad x_3 \leftrightarrow x_2 \wedge r \qquad x_4 \leftrightarrow x_3 \rightarrow x_1$$

Now we can express ϕ as the following:

nnf $\longrightarrow$ cnf cont.

The transformation is done by the Tseytin transformation (from 1968).
Example taken from Wikipedia.

Example 10.2.5

Let $\phi = ((p \lor q) \land r) \to (\neg s)$. We introduce new auxiliary variables for all subformulae:

$$x_1 \leftrightarrow \neg s \qquad x_2 \leftrightarrow p \lor q \qquad x_3 \leftrightarrow x_2 \land r \qquad x_4 \leftrightarrow x_3 \to x_1$$

Now we can express ϕ as the following:

$$\psi = x_4 \land (x_4 \leftrightarrow x_3 \to x_1) \land (x_3 \leftrightarrow x_2 \land r) \land (x_2 \leftrightarrow p \lor q) \land (x_1 \leftrightarrow \neg s)$$

Each disjunct can be turned (in polytime) into a cnf, e.g.,

nnf $\longrightarrow$ cnf cont.

The transformation is done by the Tseytin transformation (from 1968).
Example taken from Wikipedia.

Example 10.2.5

Let $\phi = ((p \lor q) \land r) \to (\neg s)$. We introduce new auxiliary variables for all subformulae:

$$x_1 \leftrightarrow \neg s \qquad x_2 \leftrightarrow p \lor q \qquad x_3 \leftrightarrow x_2 \land r \qquad x_4 \leftrightarrow x_3 \to x_1$$

Now we can express ϕ as the following:

$$\psi = x_4 \land (x_4 \leftrightarrow x_3 \to x_1) \land (x_3 \leftrightarrow x_2 \land r) \land (x_2 \leftrightarrow p \lor q) \land (x_1 \leftrightarrow \neg s)$$

Each disjunct can be turned (in polytime) into a cnf, e.g.,

$$\begin{aligned} x_2 \leftrightarrow (p \lor q) &\equiv x_2 \to (p \lor q) \land ((p \lor q) \to x_2) \\ &\equiv (\neg x_2 \lor p \lor q) \land (\neg(p \lor q) \lor x_2) \\ &\equiv (\neg x_2 \lor p \lor q) \land ((\neg p \land \neg q) \lor x_2) \\ &\equiv (\neg x_2 \lor p \lor q) \land (\neg p \lor x_2) \land (\neg q \lor x_2) \end{aligned}$$

Conclusion

We proved that *CNFSAT* is **NP**-hard!

We reduced: $\mathit{SAT} \leq_{\mathbf{P}} \mathit{nnf} \leq_{\mathbf{P}} \mathit{CNFSAT}$

Since *CNFSAT* is clearly in **NP** as well, it's **NP**-complete.

3SAT

3SAT is a special case of *CNFSAT*.

$$\mathit{3SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf formula } \}$$

where a Boolean formula is in 3cnf (for 3 literal conjunctive normal form) if it is (also) generated by the grammar

$$\phi \to (c) \mid (c) \wedge \phi \qquad\qquad c \to \ell \vee \ell \vee \ell$$
$$\ell \to p \mid \neg p \qquad\qquad p \to x \mid y \mid \ldots$$

Intuitively, a 3cnf is simply a conjunction of disjunctions of size exactly 3.

3SAT

3SAT is a special case of *CNFSAT*.

$$\mathit{3SAT} = \{\ \langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf formula } \}$$

where a Boolean formula is in 3cnf (for 3 literal conjunctive normal form) if it is (also) generated by the grammar

$$\phi \to (c) \mid (c) \wedge \phi \qquad\qquad c \to \ell \vee \ell \vee \ell$$
$$\ell \to p \mid \neg p \qquad\qquad p \to x \mid y \mid \ \ldots$$

Intuitively, a 3cnf is simply a conjunction of disjunctions of size exactly 3.

Example 10.3.1

$(x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee \neg y \vee \neg z)$ is a 3cnf for the Boolean formula x. (You can verify this by applying simplification rules or constructing a truth table.)

3SAT is **NP**-Complete

Proof.

Clearly *3SAT* is in **NP** because we can use the same certificate for ϕ in 3cnf as we would for the same ϕ in *SAT* (or *CNFSAT*). (Guess and verify.)

We **P**-reduce from *CNFSAT* to *3SAT*, by translating arbitrary clauses into clauses with exactly three literals. (We do this on the next slides.) □

Proof: *3SAT* is **NP**-hard

How to transform a cnf $\phi = \bigwedge_{i=1}^{n} c_i$ into an equisatisfiable 3cnf?

We transform each clause $c_i = \bigvee_{j=1}^{k_i} \ell_{i,j}$ depending on the number k_i of literals in it.
E.g., $c_2 = l_{2,1} \vee l_{2,2} \vee l_{2,3} \vee l_{2,4}$ with $k_2 = 4$. We omit subscript i! $c = l_1 \vee l_2 \vee l_3 \vee l_4$.

Proof: *3SAT* is **NP**-hard

How to transform a cnf $\phi = \bigwedge_{i=1}^{n} c_i$ into an equisatisfiable 3cnf?

We transform each clause $c_i = \bigvee_{j=1}^{k_i} \ell_{i,j}$ depending on the number k_i of literals in it.
E.g., $c_2 = l_{2,1} \vee l_{2,2} \vee l_{2,3} \vee l_{2,4}$ with $k_2 = 4$. We omit subscript i! $c = l_1 \vee l_2 \vee l_3 \vee l_4$.

Case $k = 1$ (ℓ_1) is replaced by

$$(\ell_1 \vee u \vee v) \wedge (\ell_1 \vee u \vee \neg v) \wedge (\ell_1 \vee \neg u \vee v) \wedge (\ell_1 \vee \neg u \vee \neg v)$$

for some fresh propositions u, v.

Proof: *3SAT* is **NP**-hard

How to transform a cnf $\phi = \bigwedge_{i=1}^{n} c_i$ into an equisatisfiable 3cnf?

We transform each clause $c_i = \bigvee_{j=1}^{k_i} \ell_{i,j}$ depending on the number k_i of literals in it.
E.g., $c_2 = l_{2,1} \vee l_{2,2} \vee l_{2,3} \vee l_{2,4}$ with $k_2 = 4$. We omit subscript i! $c = l_1 \vee l_2 \vee l_3 \vee l_4$.

Case $k = 1$ (ℓ_1) is replaced by

$$(\ell_1 \vee u \vee v) \wedge (\ell_1 \vee u \vee \neg v) \wedge (\ell_1 \vee \neg u \vee v) \wedge (\ell_1 \vee \neg u \vee \neg v)$$

for some fresh propositions u, v.

Case $k = 2$ $(\ell_1 \vee \ell_2)$ is replaced by

$$(\ell_1 \vee \ell_2 \vee u) \wedge (\ell_1 \vee \ell_2 \vee \neg u)$$

for some fresh proposition u.

Proof: *3SAT* is **NP**-hard

How to transform a cnf $\phi = \bigwedge_{i=1}^{n} c_i$ into an equisatisfiable 3cnf?

We transform each clause $c_i = \bigvee_{j=1}^{k_i} \ell_{i,j}$ depending on the number k_i of literals in it.
E.g., $c_2 = l_{2,1} \vee l_{2,2} \vee l_{2,3} \vee l_{2,4}$ with $k_2 = 4$. We omit subscript i! $c = l_1 \vee l_2 \vee l_3 \vee l_4$.

Case $k = 1$ (ℓ_1) is replaced by

$$(\ell_1 \vee u \vee v) \wedge (\ell_1 \vee u \vee \neg v) \wedge (\ell_1 \vee \neg u \vee v) \wedge (\ell_1 \vee \neg u \vee \neg v)$$

for some fresh propositions u, v.

Case $k = 2$ $(\ell_1 \vee \ell_2)$ is replaced by

$$(\ell_1 \vee \ell_2 \vee u) \wedge (\ell_1 \vee \ell_2 \vee \neg u)$$

for some fresh proposition u.

Case $k = 3$ is 3cnf already.

Proof: *3SAT* is **NP**-hard

How to transform a cnf $\phi = \bigwedge_{i=1}^{n} c_i$ into an equisatisfiable 3cnf?

We transform each clause $c_i = \bigvee_{j=1}^{k_i} \ell_{i,j}$ depending on the number k_i of literals in it.
E.g., $c_2 = l_{2,1} \vee l_{2,2} \vee l_{2,3} \vee l_{2,4}$ with $k_2 = 4$. We omit subscript i! $c = l_1 \vee l_2 \vee l_3 \vee l_4$.

Case $k = 1$ (ℓ_1) is replaced by

$$(\ell_1 \vee u \vee v) \wedge (\ell_1 \vee u \vee \neg v) \wedge (\ell_1 \vee \neg u \vee v) \wedge (\ell_1 \vee \neg u \vee \neg v)$$

for some fresh propositions u, v.

Case $k = 2$ $(\ell_1 \vee \ell_2)$ is replaced by

$$(\ell_1 \vee \ell_2 \vee u) \wedge (\ell_1 \vee \ell_2 \vee \neg u)$$

for some fresh proposition u.

Case $k = 3$ is 3cnf already.

Case $k > 3$ $(\bigvee_{j=1}^{k} \ell_j)$. On the next slide!

Proof: *3SAT* is **NP**-hard

Case $k > 3$, $(\bigvee_{j=1}^{k} \ell_j)$ is replaced by

$$(\ell_1 \vee \ell_2 \vee u_1) \wedge \bigwedge_{j=1}^{k-4} (\ell_{j+2} \vee \neg u_j \vee u_{j+1}) \wedge (\neg u_{k-3} \vee \ell_{k-1} \vee \ell_k)$$

for some $k-3$ fresh propositions $u_1, \ldots, u_{k-3}$.

Proof: *3SAT* is **NP**-hard

Case $k > 3$, $(\bigvee_{j=1}^{k} \ell_j)$ is replaced by

$$(\ell_1 \vee \ell_2 \vee u_1) \wedge \bigwedge_{j=1}^{k-4} (\ell_{j+2} \vee \neg u_j \vee u_{j+1}) \wedge (\neg u_{k-3} \vee \ell_{k-1} \vee \ell_k)$$

for some $k-3$ fresh propositions $u_1, \ldots, u_{k-3}$.

Take $l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5 \vee l_6 \vee l_7$. So $k = 7$ and $k - 3 = 4$. We can write this as:

$$\begin{aligned}
&(l_1 \vee l_2 \vee u_1) \wedge \\
&\quad (l_3 \vee \neg u_1 \vee u_2) \wedge \\
&\quad (l_4 \vee \neg u_2 \vee u_3) \wedge \\
&\quad (l_5 \vee \neg u_3 \vee u_4) \wedge \\
&(\neg u_4 \vee l_6 \vee l_7)
\end{aligned}$$

You can see that you can always pick the new propositions in a way to make all disjuncts true, no matter which literal is supposed to get true. E.g., if l_4 is true, we set u_1, u_2, u_4 true. Likewise, they don't help us making the formula true unless at least one of the l_i are true. (Check what happens if all l_i are false.)

CLIQUE is **NP**-Complete

Let $CLIQUE = \left\{ \langle G, k \rangle \;\middle|\; \begin{array}{l} G \text{ is undirected graph} \\ \text{with } k\text{-clique} \end{array} \right\}$

We show **NP**-completeness on the whiteboard. (Alban did that on Tuesday.)

HAMPATH is **NP**-Complete

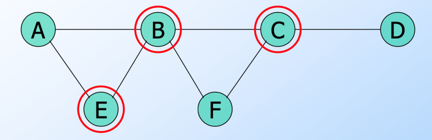
Recall that $\mathit{HAMPATH} = \left\{ \langle G, s, t \rangle \;\middle|\; \begin{array}{l} \text{Directed graph } G \text{ has a} \\ \text{Hamiltonian path from } s \text{ to } t \end{array} \right\}$

We already know that *HAMPATH* is in **NP**. We show **NP**-hardness by proving *3SAT* $\leq_{\mathbf{P}}$ *HAMPATH* on the whiteboard. (Alban did that on Tuesday.)

Node Cover

Given an undirected graph G, a node cover of G is a set C of vertices such that:

- for every edge (v_1, v_2) in the graph, at least one of v_1 or v_2 is in C. In other words: The node cover covers all edges of the graph.
- In the next example, the nodes marked in red are a node cover of the graph.



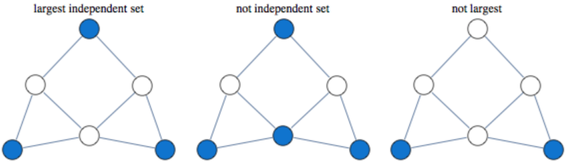
The Node Cover Problem is the problem of deciding whether a graph G has a node cover with k or fewer nodes:

$$NC = \{\langle G, k\rangle \mid G \text{ has node cover of size } \leq k\}$$

Independent Set

Given an undirected graph G, an independent set of G is a set I of vertices such that:

- no to vertices v_1 and $v_2 \in I$ are connected by an edge.

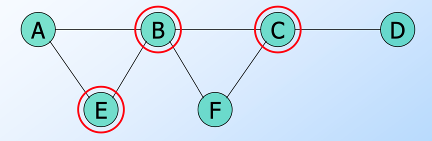


The Independent Set Problem is the problem of deciding whether a graph G has an independent set with k or more nodes:

$$IS = \{\langle G, k\rangle \mid G \text{ has independent set of size } \geq k\}$$

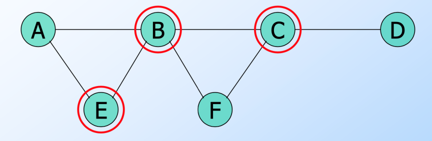
Node Cover vs. Independent Set

Q. How are node cover and independent set related?



Node Cover vs. Independent Set

Q. How are node cover and independent set related?



A. The complement of a node cover is an independent set. (See next slide.)

Node Cover vs. Independent Set II

Theorem 10.6.1

A graph G with $|V| = n$ vertices has a node cover C of size $|C| = k$ iff it has an independent set of size $n - k$. (Both problems are polytime-reducible to each other.)

Node Cover vs. Independent Set II

Theorem 10.6.1

A graph G with $|V| = n$ vertices has a node cover C of size $|C| = k$ iff it has an independent set of size $n - k$. (Both problems are polytime-reducible to each other.)

Proof.

Let G be a graph with n nodes. Let $0 \leq k \leq n$.

Claim: C is a node cover of G iff $V \setminus C$ is an independent set.

Node Cover vs. Independent Set II

Theorem 10.6.1

A graph G with $|V| = n$ vertices has a node cover C of size $|C| = k$ iff it has an independent set of size $n - k$. (Both problems are polytime-reducible to each other.)

Proof.

Let G be a graph with n nodes. Let $0 \leq k \leq n$.

Claim: C is a node cover of G iff $V \setminus C$ is an independent set.

"$\Rightarrow$" C is a node cover of G. Let $v_1, v_2 \in V \setminus C$. Show that there is no edge between v_1 and v_2. Assume there is! Then, because C is a node cover, we have $v_1 \in C$ or $v_2 \in C$. Contradiction as $v_1, v_2 \in V \setminus C$. Thus, there is no edge between v_1 and v_2 and therefore $V \setminus C$ is an independent set.

Node Cover vs. Independent Set II

Theorem 10.6.1

A graph G with $|V| = n$ vertices has a node cover C of size $|C| = k$ iff it has an independent set of size $n - k$. (Both problems are polytime-reducible to each other.)

Proof.

Let G be a graph with n nodes. Let $0 \leq k \leq n$.

Claim: C is a node cover of G iff $V \setminus C$ is an independent set.

"$\Rightarrow$" C is a node cover of G. Let $v_1, v_2 \in V \setminus C$. Show that there is no edge between v_1 and v_2. Assume there is! Then, because C is a node cover, we have $v_1 \in C$ or $v_2 \in C$. Contradiction as $v_1, v_2 \in V \setminus C$. Thus, there is no edge between v_1 and v_2 and therefore $V \setminus C$ is an independent set.

"$\not\Rightarrow$" C is not a node cover of G. Thus there is an edge (v_1, v_2), such that neither of these nodes are in C, $v_1, v_2 \notin C$. But then $v_1, v_2 \in V \setminus C$. Therefore $V \setminus C$ is not an independent set. $\square$

On the **NP**-completeness of these Problems

So far we've shown that both problems are equivalent, so how hard are they?

in **NP** Both problems are in **NP**: We can guess the respective set of nodes and check the required property. The number of guessed nodes is polytime-bounded in the input, and the property verification can also be done in poly-time.

On the **NP**-completeness of these Problems

So far we've shown that both problems are equivalent, so how hard are they?

<u>in **NP**</u> Both problems are in **NP**: We can guess the respective set of nodes and check the required property. The number of guessed nodes is polytime-bounded in the input, and the property verification can also be done in poly-time.

<u>**NP** hard</u> Since we saw that both problems are essentially the same, and once can be turned into the other just by a simple computation, we can choose for which we show hardness! Completeness then follows for both.

We show hardness for Node Cover.

On the **NP**-completeness of these Problems

So far we've shown that both problems are equivalent, so how hard are they?

in NP Both problems are in **NP**: We can guess the respective set of nodes and check the required property. The number of guessed nodes is polytime-bounded in the input, and the property verification can also be done in poly-time.

NP hard Since we saw that both problems are essentially the same, and once can be turned into the other just by a simple computation, we can choose for which we show hardness! Completeness then follows for both.

We show hardness for Node Cover.

Theorem 10.6.2

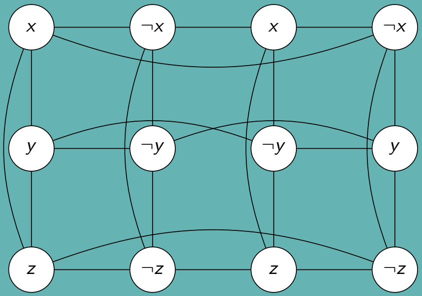
Node Cover is **NP***-hard.*

NP-hardness of Node Cover

Proof.

We reduce 3SAT to Node Cover.

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



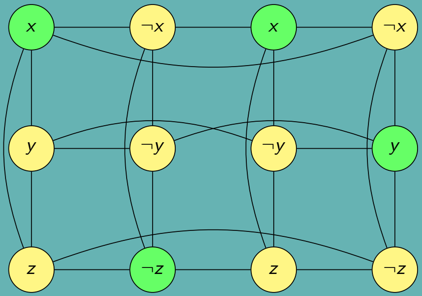
- We have one column per clause.
- Vertically, we connect all nodes within one column.
- Horizontally, we connect all contradictory nodes.
- **We claim:** ϕ is satisfiable iff G has a node cover of size $k = 2n$, where $n = 4$ is the number of clauses (select two from each column). The non-selected ones encode the literal that makes the respective clause true.

NP-hardness of Node Cover

Proof.

We reduce 3SAT to Node Cover.

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



- We have one column per clause.
- Vertically, we connect all nodes within one column.
- Horizontally, we connect all contradictory nodes.
- **We claim:** ϕ is satisfiable iff G has a node cover of size $k = 2n$, where $n = 4$ is the number of clauses (select two from each column). The non-selected ones encode the literal that makes the respective clause true.

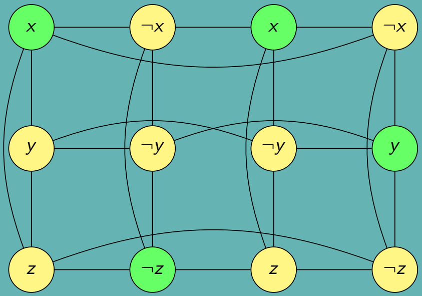
For example, $\pi(x) = \top, \pi(y) = \top, \pi(z) = \bot$ makes the formula true.

NP-hardness of Node Cover

Proof.

We reduce 3SAT to Node Cover.

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



- We have one column per clause.
- Vertically, we connect all nodes within one column.
- Horizontally, we connect all contradictory nodes.
- **We claim:** ϕ is satisfiable iff G has a node cover of size $k = 2n$, where $n = 4$ is the number of clauses (select two from each column). The non-selected ones encode the literal that makes the respective clause true.

For example, $\pi(x) = \top, \pi(y) = \top, \pi(z) = \bot$ makes the formula true.

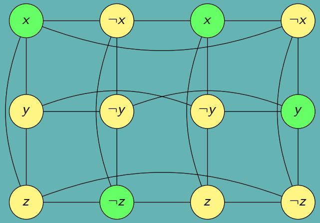
Now we still need to show this claim!

NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Rightarrow$").

Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$ we can select literal l_i of ϕ_i, s.t. $\phi \models l_i$. In our example: Let $l_1, \ldots, l_4$ be the green nodes.



NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Rightarrow$").

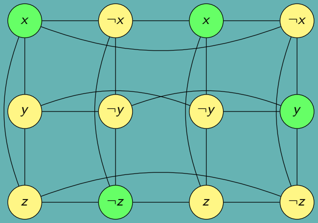
Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$ we can select literal l_i of ϕ_i, s.t. $\phi \models l_i$. In our example: Let $l_1, \ldots, l_4$ be the green nodes.

Now define the complement of these nodes as the node cover C (the yellow nodes) and show desired properties, i.e., that for each edge (v_1, v_2), at least v_1 or v_2 is in C.

vertical

horizontal



NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Rightarrow$").

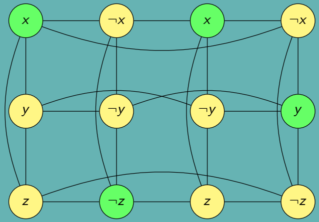
Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$ we can select literal l_i of ϕ_i, s.t. $\phi \models l_i$. In our example: Let $l_1, \ldots, l_4$ be the green nodes.

Now define the complement of these nodes as the node cover C (the yellow nodes) and show desired properties, i.e., that for each edge (v_1, v_2), at least v_1 or v_2 is in C.

vertical Selecting two nodes will *always* cover all edges.

horizontal



NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Rightarrow$").

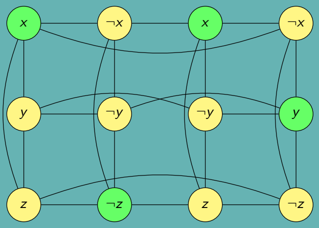
Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$ we can select literal l_i of ϕ_i, s.t. $\phi \models l_i$. In our example: Let $l_1, \ldots, l_4$ be the green nodes.

Now define the complement of these nodes as the node cover C (the yellow nodes) and show desired properties, i.e., that for each edge (v_1, v_2), at least v_1 or v_2 is in C.

vertical Selecting two nodes will *always* cover all edges.

horizontal These edges are always between a variable and its negation. So the only way to *not* have each edge covered is to have both literals selected (green), which is impossible since we can't make l_i and $\neg l_i$ true.



NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Rightarrow$").

Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

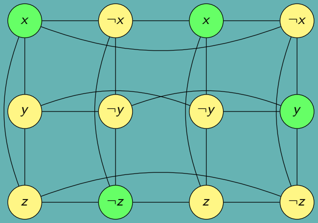
Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$ we can select literal l_i of ϕ_i, s.t. $\phi \models l_i$. In our example: Let $l_1, \ldots, l_4$ be the green nodes.

Now define the complement of these nodes as the node cover C (the yellow nodes) and show desired properties, i.e., that for each edge (v_1, v_2), at least v_1 or v_2 is in C.

vertical Selecting two nodes will *always* cover all edges.

horizontal These edges are always between a variable and its negation. So the only way to *not* have each edge covered is to have both literals selected (green), which is impossible since we can't make l_i and $\neg l_i$ true.

Q. Why did we need the vertical edges, then?
They seem apparently don't impose a constraint...



NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Rightarrow$").

Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$ we can select literal l_i of ϕ_i, s.t. $\phi \models l_i$. In our example: Let $l_1, \ldots, l_4$ be the green nodes.

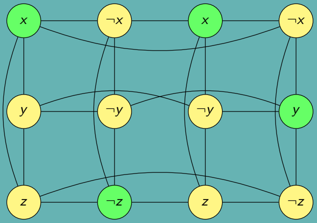
Now define the complement of these nodes as the node cover C (the yellow nodes) and show desired properties, i.e., that for each edge (v_1, v_2), at least v_1 or v_2 is in C.

vertical Selecting two nodes will *always* cover all edges.

horizontal These edges are always between a variable and its negation. So the only way to *not* have each edge covered is to have both literals selected (green), which is impossible since we can't make l_i and $\neg l_i$ true.

Q. Why did we need the vertical edges, then?
They seem apparently don't impose a constraint...

A. They did! They forced us to select a (green) literal.

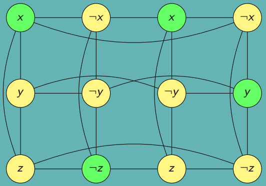


NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Leftarrow$").

Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Define assignment π, such that π makes a literal true if it's not in the node cover.



□

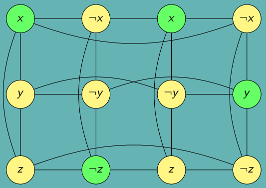
NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Leftarrow$").

Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Define assignment π, such that π makes a literal true if it's not in the node cover.

Notice that node cover of size $k = 2n$ needs to select precisely 2 elements from each column: because if it doesn't we can always find an edge that's not covered.



$\square$

NP-hardness of Node Cover (cont'd)

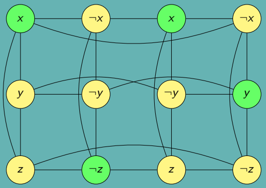
Proof. (Reduction, "$\Leftarrow$").

Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Define assignment π, such that π makes a literal true if it's not in the node cover.

Notice that node cover of size $k = 2n$ needs to select precisely 2 elements from each column: because if it doesn't we can always find an edge that's not covered.

So, why does any node cover (with two yellow nodes in each column) encode an assignment π that makes the formula true?



$\square$

NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Leftarrow$").

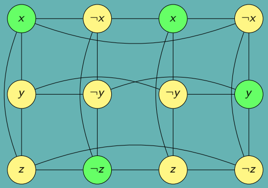
Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Define assignment π, such that π makes a literal true if it's not in the node cover.

Notice that node cover of size $k = 2n$ needs to select precisely 2 elements from each column: because if it doesn't we can always find an edge that's not covered.

So, why does any node cover (with two yellow nodes in each column) encode an assignment π that makes the formula true?

- Again, all nodes *not* in that cover give the witness for making the respective clause true.
- Thus, each clause already has a witness making it true!



□

NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Leftarrow$").

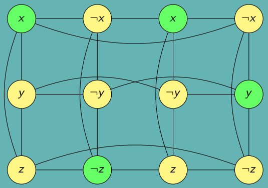
Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Define assignment π, such that π makes a literal true if it's not in the node cover.

Notice that node cover of size $k = 2n$ needs to select precisely 2 elements from each column: because if it doesn't we can always find an edge that's not covered.

So, why does any node cover (with two yellow nodes in each column) encode an assignment π that makes the formula true?

- Again, all nodes *not* in that cover give the witness for making the respective clause true.
- Thus, each clause already has a witness making it true!



So what could still go wrong?

$\square$

NP-hardness of Node Cover (cont'd)

Proof. (Reduction, "$\Leftarrow$").

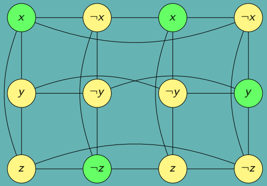
Recall: ϕ is satisfiable iff G has a node cover of size $k = 2n$

Define assignment π, such that π makes a literal true if it's not in the node cover.

Notice that node cover of size $k = 2n$ needs to select precisely 2 elements from each column: because if it doesn't we can always find an edge that's not covered.

So, why does any node cover (with two yellow nodes in each column) encode an assignment π that makes the formula true?

- Again, all nodes *not* in that cover give the witness for making the respective clause true.
- Thus, each clause already has a witness making it true!



So what could still go wrong? We need consistent assignments!

- I.e., don't make some literal l_i true and false, $\pi(l_i) = \pi(\neg l_i) = \top$.
- This can't happen! They all share a (horizontal) edge, so selecting both for π (green) would exclude both for the node cover – leaving a non-covered edge.

$\square$