COMP3630 / COMP6363

week 8: Classes P, NP, and co-NP

This Lecture Covers Chapter 10 and 11 of HMU: *Intractable Problems* and *Additional Classes of Problems*

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, 2025

Content of this Chapter

- Problems in **P**
- Class **NP**
- SAT is **NP**-hard // patient zero!
- Problems that are **NP**-complete
- Class **Co-NP**

Additional Reading: Chapters 10 and 11 of HMU.

Problems in **P**

Context-Free Languages

Theorem w8.1

Every CFL is in **P**.

First, what does that mean? When is a language in **P**?

› $L \in \mathbf{P}$ iff there's a total TM M with $L(M) = L$ that runs in polytime.

Context-Free Languages

Theorem w8.1

Every CFL is in **P**.

First, what does that mean? When is a language in **P**?

- $L \in \mathbf{P}$ iff there's a total TM M with $L(M) = L$ that runs in polytime.
- Note that a TM can represent everything from this course: DFAs, NFAs, grammars, ... Thus, we pick the most suitable representation for which we know a polytime procedure and assume it's given in terms of a TM.

Context-Free Languages

Theorem w8.1

Every CFL is in **P**.

First, what does that mean? When is a language in **P**?

- $L \in \mathbf{P}$ iff there's a total TM M with $L(M) = L$ that runs in polytime.
- Note that a TM can represent everything from this course: DFAs, NFAs, grammars, ... Thus, we pick the most suitable representation for which we know a polytime procedure and assume it's given in terms of a TM.

Proof.

- Let L be context-free.
- We thus know that there exists a CFG G in Chomsky Normal Form (CNF).
- Let G be given. Now run CYK (implemented within our TM) on the input w, taking $\mathcal{O}(|w|^3)$ time. □

Context-Free Languages

Theorem w8.1

Every CFL is in **P**.

First, what does that mean? When is a language in **P**?

- $L \in \mathbf{P}$ iff there's a total TM M with $L(M) = L$ that runs in polytime.
- Note that a TM can represent everything from this course: DFAs, NFAs, grammars, ... Thus, we pick the most suitable representation for which we know a polytime procedure and assume it's given in terms of a TM.

Proof.

- Let L be context-free.
- We thus know that there exists a CFG G in Chomsky Normal Form (CNF).
- Let G be given. Now run CYK (implemented within our TM) on the input w, taking $\mathcal{O}(|w|^3)$ time. □

Note that this proof would even be correct if converting a CFG G into CNF would take exponential time! Thus, this theorem does not say that $\{(A, w) \mid w \in L(A)\}$ is polytime decidable for any CFL or automaton A where $L(A)$ is context-free.

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

Follows from the last theorem: every CFL is in **P** □

Clear?

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

Follows from the last theorem: every CFL is in **P** □

Clear?

It shouldn't, as the proof is wrong ! Why?

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

Follows from the last theorem: every CFL is in **P** □

Clear?

It shouldn't, as the proof is wrong ! Why? We never proved that L is context-free! $L(G)$ is context-free for any CFG G, but here your language isn't $L(G)$, but L from above!

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

- Since G is a CFG, we know that a CFG G' in CNF exists.
- Let M be the TM that implements CYK on G'.
- Run it on w in polytime. Accept or reject accordingly.

□

Correct now?

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

- Since G is a CFG, we know that a CFG G' in CNF exists.
- Let M be the TM that implements CYK on G'.
- Run it on w in polytime. Accept or reject accordingly.

□

Correct now?

No, wrong again ! Why?

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

- Since G is a CFG, we know that a CFG G' in CNF exists.
- Let M be the TM that implements CYK on G'.
- Run it on w in polytime. Accept or reject accordingly.

□

Correct now?

No, wrong again ! Why? Because you must provide a TM that receives G and w as input and then run in polytime to make the decision. Here you provide a TM for G', which only works for G/G', but not for arbitrary input words (G, w).

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

- Since G is a CFG, we know that we can transform G into a CFG G' in CNF in polynomial time.
- Let M be the TM that first does the above normalization and then implements CYK on the resulting CFG G'.
- Run it on w in polytime. Accept or reject accordingly.

□

Correct now?

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

- Since G is a CFG, we know that we can transform G into a CFG G' in CNF in polynomial time.
- Let M be the TM that first does the above normalization and then implements CYK on the resulting CFG G'.
- Run it on w in polytime. Accept or reject accordingly.

□

Correct now?

Still wrong ! Why?

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

- Since G is a CFG, we know that we can transform G into a CFG G' in CNF in polynomial time.
- Let M be the TM that first does the above normalization and then implements CYK on the resulting CFG G'.
- Run it on w in polytime. Accept or reject accordingly.

□

Correct now?

Still wrong ! Why? Establishing CNF takes exponential time as taught due to elimination of nullable variables, which results in exponentially many new rules.

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

- We want to turn CFG G into CFG G' in CNF in polynomial time. The only step that takes exponential time is the elimination of nullable variables.
- So, we first normalize G so that all production rules have fixed size, e.g., 2 (like in step 4). Now, we can eliminate nullable variables in polynomial time.
- Let M be the TM that first does the above normalization and then implements CYK on the resulting CFG G'.
- Run it on w in polytime. Accept or reject accordingly.

□

Correct now?

More Problems in **P**, cont'd

Theorem w8.2

Deciding $L = \{(G, w) \mid G \text{ is CFG and } w \in L(G)\}$ *is in* **P**.

Proof.

- We want to turn CFG G into CFG G' in CNF in polynomial time. The only step that takes exponential time is the elimination of nullable variables.
- So, we first normalize G so that all production rules have fixed size, e.g., 2 (like in step 4). Now, we can eliminate nullable variables in polynomial time.
- Let M be the TM that first does the above normalization and then implements CYK on the resulting CFG G'.
- Run it on w in polytime. Accept or reject accordingly.

□

Correct now?

Yes. :)

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

□

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

› If n is composite, then it can be written as $n = a \cdot b$ with $1 < a \leq b < n$.

□

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

- If n is composite, then it can be written as $n = a \cdot b$ with $1 < a \leq b < n$.
- If both a and b were greater than $\sqrt{n}$, then $a \cdot b > n$ – contradiction.

□

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

- If n is composite, then it can be written as $n = a \cdot b$ with $1 < a \leq b < n$.
- If both a and b were greater than $\sqrt{n}$, then $a \cdot b > n$ – contradiction.
- So, at least one factor of any composite number n must be $\leq \sqrt{n}$.

□

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

- If n is composite, then it can be written as $n = a \cdot b$ with $1 < a \leq b < n$.
- If both a and b were greater than $\sqrt{n}$, then $a \cdot b > n$ – contradiction.
- So, at least one factor of any composite number n must be $\leq \sqrt{n}$.
- Thus, checking all numbers i from 2 to $\lfloor \sqrt{n} \rfloor$ suffices. For each, we test whether $n \bmod i = 0$. If all are "no", return "yes" (prime), otherwise "no". This takes $\mathcal{O}(\sqrt{n}) = \mathcal{O}(n^{\frac{1}{2}})$. We thus get **P** membership. □

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

- If n is composite, then it can be written as $n = a \cdot b$ with $1 < a \leq b < n$.
- If both a and b were greater than $\sqrt{n}$, then $a \cdot b > n$ – contradiction.
- So, at least one factor of any composite number n must be $\leq \sqrt{n}$.
- Thus, checking all numbers i from 2 to $\lfloor \sqrt{n} \rfloor$ suffices. For each, we test whether $n \bmod i = 0$. If all are "no", return "yes" (prime), otherwise "no". This takes $\mathcal{O}(\sqrt{n}) = \mathcal{O}(n^{\frac{1}{2}})$. We thus get **P** membership. □

Right? :)

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

- If n is composite, then it can be written as $n = a \cdot b$ with $1 < a \leq b < n$.
- If both a and b were greater than $\sqrt{n}$, then $a \cdot b > n$ – contradiction.
- So, at least one factor of any composite number n must be $\leq \sqrt{n}$.
- Thus, checking all numbers i from 2 to $\lfloor \sqrt{n} \rfloor$ suffices. For each, we test whether $n \bmod i = 0$. If all are "no", return "yes" (prime), otherwise "no". This takes $\mathcal{O}(\sqrt{n}) = \mathcal{O}(n^{\frac{1}{2}})$. We thus get **P** membership. □

Right? :) No! Two errors:

- $n \bmod i = 0$ does not take constant time, but $\mathcal{O}(|n|^2)$. (Still poly!)

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

- If n is composite, then it can be written as $n = a \cdot b$ with $1 < a \leq b < n$.
- If both a and b were greater than $\sqrt{n}$, then $a \cdot b > n$ – contradiction.
- So, at least one factor of any composite number n must be $\leq \sqrt{n}$.
- Thus, checking all numbers i from 2 to $\lfloor \sqrt{n} \rfloor$ suffices. For each, we test whether $n \bmod i = 0$. If all are "no", return "yes" (prime), otherwise "no". This takes $\mathcal{O}(\sqrt{n}) = \mathcal{O}(n^{\frac{1}{2}})$. We thus get **P** membership. □

Right? :) No! Two errors:

- $n \bmod i = 0$ does not take constant time, but $\mathcal{O}(|n|^2)$. (Still poly!)
- $\mathcal{O}(\sqrt{n})$ must be measured in terms of the input size $|n|$.
  Since $|n| = \log(n)$, i.e., $n = 2^{|n|}$, we get: $\mathcal{O}(\sqrt{n}) = \mathcal{O}(n^{\frac{1}{2}}) = \mathcal{O}(2^{\frac{|n|}{2}})$.

→ In total, we get $\mathcal{O}(2^{\frac{1}{2}|n|}|n|^2)$. Hence, this algorithm proves **EXPTIME** membership!

More Problems in **P**, cont'd II

Theorem w8.3

Deciding PRIME $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$ *is in* **P**.

Proof.

- If n is composite, then it can be written as $n = a \cdot b$ with $1 < a \leq b < n$.
- If both a and b were greater than $\sqrt{n}$, then $a \cdot b > n$ – contradiction.
- So, at least one factor of any composite number n must be $\leq \sqrt{n}$.
- Thus, checking all numbers i from 2 to $\lfloor \sqrt{n} \rfloor$ suffices. For each, we test whether $n \bmod i = 0$. If all are "no", return "yes" (prime), otherwise "no". This takes $\mathcal{O}(\sqrt{n}) = \mathcal{O}(n^{\frac{1}{2}})$. We thus get **P** membership. □

Right? :) No! Two errors:

- $n \bmod i = 0$ does not take constant time, but $\mathcal{O}(|n|^2)$. (Still poly!)
- $\mathcal{O}(\sqrt{n})$ must be measured in terms of the input size $|n|$.
  Since $|n| = \log(n)$, i.e., $n = 2^{|n|}$, we get: $\mathcal{O}(\sqrt{n}) = \mathcal{O}(n^{\frac{1}{2}}) = \mathcal{O}(2^{\frac{|n|}{2}})$.

→ In total, we get $\mathcal{O}(2^{\frac{1}{2}|n|}|n|^2)$. Hence, this algorithm proves **EXPTIME** membership!

But *PRIME* is indeed in **P**, proved in 2002. (Proof skipped.)

Class **NP**

Conversion of non-det. TM into det. TM

Theorem w8.1

Let $t : \mathbb{N} \to \mathbb{N}$. For every t-time non-deterministic TM M with $L(M) = L$, there is a $2^{\mathcal{O}(t(n))}$-time deterministic TM M' with $L(M') = L$.

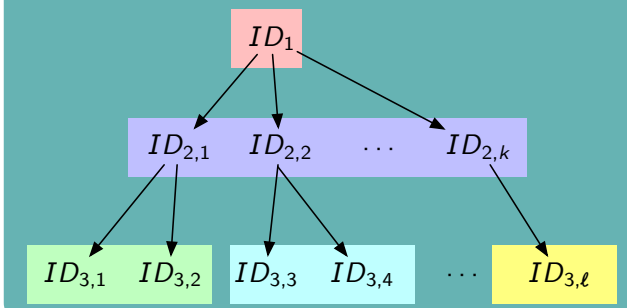
Conversion of non-det. TM into det. TM

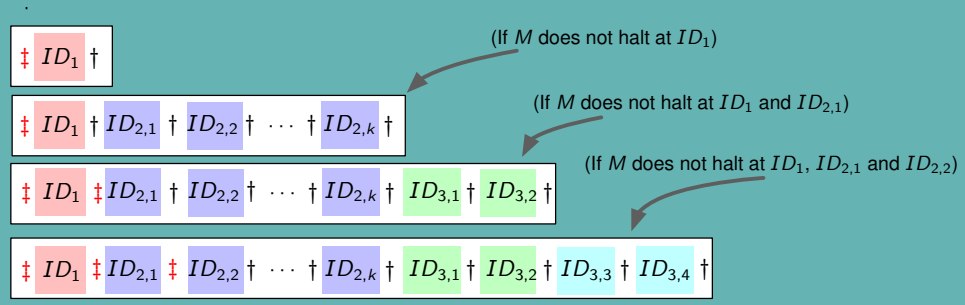
Theorem w8.1

Let $t : \mathbb{N} \to \mathbb{N}$. For every t-time non-deterministic TM M with $L(M) = L$, there is a $2^{\mathcal{O}(t(n))}$-time deterministic TM M' with $L(M') = L$.

Proof.

By analyzing the time complexity of the construction given to show that every non-deterministic TM has an equivalent deterministic TM. (Cf. week 5)





□

Conversion of non-det. TM into det. TM

Theorem w8.1

Let $t : \mathbb{N} \to \mathbb{N}$. For every t-time non-deterministic TM M with $L(M) = L$, there is a $2^{\mathcal{O}(t(n))}$-time deterministic TM M' with $L(M') = L$.

Proof.

By analyzing the time complexity of the construction given to show that every non-deterministic TM has an equivalent deterministic TM.

› For inputs of length n the computation tree of M has depth at most $t(n)$.

□

Conversion of non-det. TM into det. TM

Theorem w8.1

Let $t : \mathbb{N} \to \mathbb{N}$. For every t-time non-deterministic TM M with $L(M) = L$, there is a $2^{\mathcal{O}(t(n))}$-time deterministic TM M' with $L(M') = L$.

Proof.

By analyzing the time complexity of the construction given to show that every non-deterministic TM has an equivalent deterministic TM.

- For inputs of length n the computation tree of M has depth at most $t(n)$.
- Every tree node has at most b children, where $b \in \mathbb{N}$ depends on M's transition function. Thus the tree has no more than $b^{t(n)+1}$ nodes.

□

Conversion of non-det. TM into det. TM

Theorem w8.1

Let $t : \mathbb{N} \to \mathbb{N}$. For every t-time non-deterministic TM M with $L(M) = L$, there is a $2^{\mathcal{O}(t(n))}$-time deterministic TM M' with $L(M') = L$.

Proof.

By analyzing the time complexity of the construction given to show that every non-deterministic TM has an equivalent deterministic TM.

- For inputs of length n the computation tree of M has depth at most $t(n)$.
- Every tree node has at most b children, where $b \in \mathbb{N}$ depends on M's transition function. Thus the tree has no more than $b^{t(n)+1}$ nodes.
- M' may have to explore all of them, in a breadth-first fashion. Each exploration may take $\mathcal{O}(t(n))$ steps (from the root to a node).

□

Conversion of non-det. TM into det. TM

Theorem w8.1

Let $t : \mathbb{N} \to \mathbb{N}$. For every t-time non-deterministic TM M with $L(M) = L$, there is a $2^{\mathcal{O}(t(n))}$-time deterministic TM M' with $L(M') = L$.

Proof.

By analyzing the time complexity of the construction given to show that every non-deterministic TM has an equivalent deterministic TM.

- For inputs of length n the computation tree of M has depth at most $t(n)$.
- Every tree node has at most b children, where $b \in \mathbb{N}$ depends on M's transition function. Thus the tree has no more than $b^{t(n)+1}$ nodes.
- M' may have to explore all of them, in a breadth-first fashion. Each exploration may take $\mathcal{O}(t(n))$ steps (from the root to a node).
- So all explorations together may take $\mathcal{O}(t(n)) \cdot \mathcal{O}(b^{t(n)+1}) = 2^{\mathcal{O}(t(n))}$ time.

□

How to show **NP** membership

Recall that we defined **NP** as $\bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(n^k)$.
Thus, to prove **NP** membership of problem P, we had to:

- provide a non-det TM M with $L(M) = L$ that runs in $\mathcal{O}(n^k)$ time, for some $k \in \mathbb{N}$.

How to show **NP** membership

Recall that we defined **NP** as $\bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k)$.
Thus, to prove **NP** membership of problem P, we had to:

- provide a non-det TM M with $L(M) = L$ that runs in $\mathcal{O}(n^k)$ time, for some $k \in \mathbb{N}$.
- In practice, we do however deploy a two-stage process:
  - Guess a certificate c with $|c| \leq p(|w|)$.
  - Verify certificate $|c|$ in $\mathcal{O}(|c|)$.

How to show **NP** membership

Recall that we defined **NP** as $\bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k)$.
Thus, to prove **NP** membership of problem P, we had to:

- provide a non-det TM M with $L(M) = L$ that runs in $\mathcal{O}(n^k)$ time, for some $k \in \mathbb{N}$.
- In practice, we do however deploy a two-stage process:
  - Guess a certificate c with $|c| \leq p(|w|)$.
  - Verify certificate $|c|$ in $\mathcal{O}(|c|)$.
- Formally, we can say: If L is the language, say, $L = \{w \mid \mathcal{P}(w)\}$, where $\mathcal{P}$ is the property that w must possess, then, the certificate may be a representation of:
  - "the required property $\mathcal{P}$ itself", or
- For example,
  - for SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$, the certificate would be the valuation itself,

How to show **NP** membership

Recall that we defined **NP** as $\bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k)$.
Thus, to prove **NP** membership of problem P, we had to:

- provide a non-det TM M with $L(M) = L$ that runs in $\mathcal{O}(n^k)$ time, for some $k \in \mathbb{N}$.
- In practice, we do however deploy a two-stage process:
  - Guess a certificate c with $|c| \leq p(|w|)$.
  - Verify certificate $|c|$ in $\mathcal{O}(|c|)$.
- Formally, we can say: If L is the language, say, $L = \{w \mid \mathcal{P}(w)\}$, where $\mathcal{P}$ is the property that w must possess, then, the certificate may be a representation of:
  - "the required property $\mathcal{P}$ itself", or
  - any proof/argument from which it follows that w has the property $\mathcal{P}$.
- For example,
  - for SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$, the certificate would be the valuation itself,
  - but for some problems, the "property" might be exponentially long! In that case we need a more abstract certificate, as otherwise we could not be in polytime!

How to show **NP** membership

Recall that we defined **NP** as $\bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k)$.
Thus, to prove **NP** membership of problem P, we had to:

- provide a non-det TM M with $L(M) = L$ that runs in $\mathcal{O}(n^k)$ time, for some $k \in \mathbb{N}$.
- In practice, we do however deploy a two-stage process:
  - Guess a certificate c with $|c| \leq p(|w|)$.
  - Verify certificate $|c|$ in $\mathcal{O}(|c|)$.
- Formally, we can say: If L is the language, say, $L = \{w \mid \mathcal{P}(w)\}$, where $\mathcal{P}$ is the property that w must possess, then, the certificate may be a representation of:
  - "the required property $\mathcal{P}$ itself", or
  - any proof/argument from which it follows that w has the property $\mathcal{P}$.
- For example,
  - for SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$, the certificate would be the valuation itself,
  - but for some problems, the "property" might be exponentially long! In that case we need a more abstract certificate, as otherwise we could not be in polytime!
- We can conclude that **NP** is not the class of problems that possess a poly-sized "solution" (but a poly-sized "certificate").

Alternative Definition of **NP**

Definition w8.2

A verifier for a language L is a deterministic TM V, such that:
$w \in L$ iff there is a certificate c (a string), such that $(w, c) \in L(V)$

Theorem w8.3

NP *equals the set of languages that have polynomial-time verifiers with poly-sized certificates (relative to* w, $|c| \leq p(|w|)$ *for some polynomial* p*).*

Alternative Definition of **NP**

Definition w8.2

A verifier for a language L is a deterministic TM V, such that:
$w \in L$ iff there is a certificate c (a string), such that $(w, c) \in L(V)$

Theorem w8.3

NP *equals the set of languages that have polynomial-time verifiers with poly-sized certificates (relative to* w, $|c| \leq p(|w|)$ *for some polynomial* p*).*

Proof.

$\Rightarrow$: Let $L \in$ **NP**. Then, there exists a poly-time NTM M with $L = L(M)$. Then there must be a sequence of configurations, poly-length-bounded in w, which ends in an accepting state/configuration. Use this sequence as certificate: Design a deterministic verifier V, which tests whether a given sequence of configurations is valid with regard to M and ends in an accepting configuration. Accept/reject accordingly.

Alternative Definition of **NP**

Definition w8.2

A verifier for a language L is a deterministic TM V, such that:
$w \in L$ iff there is a certificate c (a string), such that $(w, c) \in L(V)$

Theorem w8.3

NP *equals the set of languages that have polynomial-time verifiers with poly-sized certificates (relative to* w, $|c| \leq p(|w|)$ *for some polynomial* p*).*

Proof.

$\Rightarrow$: Let $L \in$ **NP**. Then, there exists a poly-time NTM M with $L = L(M)$. Then there must be a sequence of configurations, poly-length-bounded in w, which ends in an accepting state/configuration. Use this sequence as certificate: Design a deterministic verifier V, which tests whether a given sequence of configurations is valid with regard to M and ends in an accepting configuration. Accept/reject accordingly.
$\Leftarrow$: For L, we know that a (deterministic) verifier V exists, and for each word $w \in L$ a certificate that's poly-length in w. Design a poly-time NTM M with $L = L(M)$ as follows. First, M non-deterministically generates all possible certificates (i.e., each run produces one certificate). Then, we switch into the second phase which implements V, which in turn verifies the certificate and accepts/rejects accordingly. Thus, $L \in$ **NP**. $\square$

SAT is **NP**-hard

Making our life easier...

So, for **NP**-completeness we need to show **NP**-hardness. For this, we had two options:

Making our life easier...

So, for **NP**-completeness we need to show **NP**-hardness. For this, we had two options:

1. Use the "text book definition", i.e., show that all problems in **NP** reduce to our problem, or
2. use the respective theorem that allows to reduce from any **NP**-hard problem.

Making our life easier...

So, for **NP**-completeness we need to show **NP**-hardness. For this, we had two options:

1. Use the "text book definition", i.e., show that all problems in **NP** reduce to our problem, or
2. use the respective theorem that allows to reduce from any **NP**-hard problem.

So, in the first case we need to show a property for all problems, in the second we only need a single reduction... What's easier? :)

Making our life easier...

So, for **NP**-completeness we need to show **NP**-hardness. For this, we had two options:

1. Use the "text book definition", i.e., show that all problems in **NP** reduce to our problem, or
2. use the respective theorem that allows to reduce from any **NP**-hard problem.

So, in the first case we need to show a property for all problems, in the second we only need a single reduction... What's easier? :)

So, we need a very first problem that's shown to be **NP**-hard – from then on we can start reducing!

For this, we will use SAT!

Boolean Formulae

Let $V = \{x, y, \ldots\}$ be a (finite) set of Boolean variables (or propositions).
A CFG to generate well-formed Boolean formulae over V is:

$$\phi \to p \mid \phi \wedge \phi \mid \neg\phi \mid (\phi)$$
$$p \to x \mid y \mid \ldots$$

We use abbreviations such as

$$\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2) \qquad \phi_1 \Rightarrow \phi_2 = \neg\phi_1 \vee \phi_2$$
$$\text{FALSE} = (x \wedge \neg x) \qquad \text{TRUE} = \neg\text{FALSE}$$

Semantics of Boolean Formulae

A Boolean formula is either $\top$ (for "true") or $\bot$ (for "false"), possibly depending on the interpretation of its propositions. Let $\mathbb{B} = \{\bot, \top\}$.

Definition w8.1

An interpretation (or assignment or valuation) of V is a function $\pi : V \longrightarrow \mathbb{B}$.
For Boolean formulae ϕ we define that π satisfies ϕ, written $\pi \models \phi$, inductively by:
Base: For each variable $x \in V$, $\pi \models x$ iff $\pi(x) = \top$.
Induction:

- $\pi \models \neg\phi$ iff $\pi \not\models \phi$.
- $\pi \models \phi_1 \wedge \phi_2$ iff both $\pi \models \phi_1$ and $\pi \models \phi_2$.
- $\pi \models (\phi)$ iff $\pi \models \phi$.

ϕ is satisfiable if there exists an interpretation π such that $\pi \models \phi$.

SAT—An **NP**-Complete Problem

Our decision problem/language is:

$$SAT = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula } \}$$

Theorem w8.2 (Cook-Levin Theorem – or: Cook's Theorem, 1971/1973)

SAT is **NP***-complete.*

Proof of $SAT \in$ NP.

If $\pi \models \phi$ we use $\langle \pi \rangle$ as certificate. (I.e., guess it and verify.) □

Proof of *SAT* is NP-hard.

The rest of this section. □

Proof of **NP**-Hardness of *SAT*

Let $L \in \mathbf{NP}$. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a deciding NTM with $L(M) = L$ and let p be a polynomial such that M takes at most $p(|w|)$ steps on any computation for any $w \in \Sigma^*$. (We know that M and p exist due to $L \in \mathbf{NP}$.)

Construct a **P**-reduction from L to *SAT*:

- Input w is turned into a Boolean formula ϕ_w that describes M's possible computations on w.
- M accepts w iff ϕ_w is satisfiable. The satisfying interpretation resolves the nondeterminism in the computation tree to arrive at an accepting branch of the computation tree.

Remains to be done: define ϕ_w.

Proof of **NP**-Hardness of *SAT* cont.

Recall that M accepts w if an $n \leq p(|w|)$ exists and a sequence of configurations $(C_i)_{0 \leq i \leq n}$ (IDs), where

1. $C_0 = q_0 w$,
2. each C_i can yield C_{i+1}, and
3. C_n is an accepting ID.
4. Note that we have at most $n+1$ IDs if the TM can make at most $n \leq p(|w|)$ steps.

ϕ_w

The Boolean formula ϕ_w shall represent all such sequences $(C_i)_{0<i\leq n}$ beginning with $q_0 w$.

$$\phi_w = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

The different sub formulae serve the following purposes:

- ϕ_{cell}: Defines all existing "cells", which encode all possible IDs.
- ϕ_{start}: Sets the initial row of these cells: TM's initial ID.
- ϕ_{move}: Enforces legal TM transitions.
- ϕ_{accept}: Enforces ending up in an accepting state.

ϕ_{cell}

. . . describes an n^2 grid using propositions $V = \{ x_{i,k,s} \mid i, k \in \{0, \ldots, n\} \wedge s \in \Sigma_\phi \}$, where $\Sigma_\phi = Q \cup \Gamma$ (recall that $B \in \Gamma$) is the "alphabet of the SAT formula" used to encode the IDs. Also recall that TM IDs contain the non-trivial tape and the state.

First, why is $i, k \in \{0, \ldots, n\}$? Why $s \in \Sigma_\phi$?

ϕ_{cell}

. . . describes an n^2 grid using propositions $V = \{ x_{i,k,s} \mid i, k \in \{0, \ldots, n\} \wedge s \in \Sigma_\phi \}$, where $\Sigma_\phi = Q \cup \Gamma$ (recall that $B \in \Gamma$) is the "alphabet of the SAT formula" used to encode the IDs. Also recall that TM IDs contain the non-trivial tape and the state.

First, why is $i, k \in \{0, \ldots, n\}$? Why $s \in \Sigma_\phi$?

- i: encodes the rows. We need one for every possible ID ($n + 1$ many!)

ϕ_{cell}

. . . describes an n^2 grid using propositions $V = \{ x_{i,k,s} \mid i, k \in \{0, \ldots, n\} \wedge s \in \Sigma_\phi \}$, where $\Sigma_\phi = Q \cup \Gamma$ (recall that $B \in \Gamma$) is the "alphabet of the SAT formula" used to encode the IDs. Also recall that TM IDs contain the non-trivial tape and the state.

First, why is $i, k \in \{0, \ldots, n\}$? Why $s \in \Sigma_\phi$?

- i: encodes the rows. We need one for every possible ID ($n + 1$ many!)
- k: encodes the columns. Each column is a possible value of an ID symbol. n symbols are the TM cells that can be reached, and one is the state.

ϕ_{cell}

. . . describes an n^2 grid using propositions $V = \{ x_{i,k,s} \mid i, k \in \{0, \ldots, n\} \wedge s \in \Sigma_\phi \}$, where $\Sigma_\phi = Q \cup \Gamma$ (recall that $B \in \Gamma$) is the "alphabet of the SAT formula" used to encode the IDs. Also recall that TM IDs contain the non-trivial tape and the state.

First, why is $i, k \in \{0, \ldots, n\}$? Why $s \in \Sigma_\phi$?

- i: encodes the rows. We need one for every possible ID ($n + 1$ many!)
- k: encodes the columns. Each column is a possible value of an ID symbol. n symbols are the TM cells that can be reached, and one is the state.
- s: The content of ID i at position k, i.e., a tape symbol or the state.

ϕ_{cell}

. . . describes an n^2 grid using propositions $V = \{ x_{i,k,s} \mid i, k \in \{0, \ldots, n\} \wedge s \in \Sigma_\phi \}$, where $\Sigma_\phi = Q \cup \Gamma$ (recall that $B \in \Gamma$) is the "alphabet of the SAT formula" used to encode the IDs. Also recall that TM IDs contain the non-trivial tape and the state.

First, why is $i, k \in \{0, \ldots, n\}$? Why $s \in \Sigma_\phi$?

- i: encodes the rows. We need one for every possible ID ($n + 1$ many!)
- k: encodes the columns. Each column is a possible value of an ID symbol. n symbols are the TM cells that can be reached, and one is the state.
- s: The content of ID i at position k, i.e., a tape symbol or the state.

$$\phi_{\text{cell}} = \bigwedge_{0 \leq i,k \leq n} \left(\left(\bigvee_{s \in \Sigma_\phi} x_{i,k,s} \right) \wedge \left(\bigwedge_{s \neq t \in \Sigma_\phi} (\neg x_{i,k,s} \vee \neg x_{i,k,t}) \right) \right)$$

Meaning: "There is exactly one symbol at each cell".

ϕ_{start}

. . . specifies that the first row of the grid contains $q_0 w$ where $w = w_1 \dots w_{|w|}$:

$$\phi_{\text{start}} = x_{0,0,q_0} \wedge \bigwedge_{1 \leq i \leq |w|} x_{0,i,w_i} \wedge \bigwedge_{|w| < i \leq n} x_{0,i,B}$$

So, the first line of our grid contains:

- the q_0 symbol in the first cell,
- followed by the symbols of our initial tape word,
- followed by the blank symbol until the end.

ϕ_{move}

. . . ensures that C_i yields C_{i+1} by describing legal 2×3 windows of cells. We need 3 cells to cover the cell on the left of the state, the state, and on its right (to enable left and right movements of the head).

$$\phi_{\text{move}} = \bigwedge_{0<i,k<n} \bigvee_{\begin{array}{|c|c|c|}\hline a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 \\ \hline\end{array}\ \text{is legal}} \left(\begin{array}{lll} x_{i,k-1,a_1} & \wedge\, x_{i,k,a_2} & \wedge\, x_{i,k+1,a_3} \wedge \\ x_{i+1,k-1,a_4} & \wedge\, x_{i+1,k,a_5} & \wedge\, x_{i+1,k+1,a_6} \end{array} \right)$$

(Some border cases are not be covered here for simplicity, e.g., i can never be zero.)
What is legal depends on the transition function δ.

ϕ_{move}

. . . ensures that C_i yields C_{i+1} by describing legal 2×3 windows of cells. We need 3 cells to cover the cell on the left of the state, the state, and on its right (to enable left and right movements of the head).

$$\phi_{\text{move}} = \bigwedge_{0<i,k<n} \bigvee_{\begin{array}{|c|c|c|}\hline a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 \\ \hline\end{array}\ \text{is legal}} \begin{pmatrix} x_{i,k-1,a_1} & \wedge\, x_{i,k,a_2} & \wedge\, x_{i,k+1,a_3} \wedge \\ x_{i+1,k-1,a_4} & \wedge\, x_{i+1,k,a_5} & \wedge\, x_{i+1,k+1,a_6} \end{pmatrix}$$

(Some border cases are not be covered here for simplicity, e.g., i can never be zero.)
What is legal depends on the transition function δ.

Example: Let the current ID be $w_1 w_2 q w_3 w_4$ (so, we have blanks before and after it). Whether we go to the left or to the right, we only need to change 3 cells!

- $w_1 w_2 q w_3 w_4$ – current ID

ϕ_{move}

. . . ensures that C_i yields C_{i+1} by describing legal 2×3 windows of cells. We need 3 cells to cover the cell on the left of the state, the state, and on its right (to enable left and right movements of the head).

$$\phi_{\text{move}} = \bigwedge_{0<i,k<n} \bigvee_{\begin{array}{|c|c|c|}\hline a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 \\ \hline\end{array}\ \text{is legal}} \left(\begin{array}{lll} x_{i,k-1,a_1} & \wedge\, x_{i,k,a_2} & \wedge\, x_{i,k+1,a_3} \wedge \\ x_{i+1,k-1,a_4} & \wedge\, x_{i+1,k,a_5} & \wedge\, x_{i+1,k+1,a_6} \end{array} \right)$$

(Some border cases are not be covered here for simplicity, e.g., i can never be zero.) What is legal depends on the transition function δ.

Example: Let the current ID be $w_1 w_2 q w_3 w_4$ (so, we have blanks before and after it). Whether we go to the left or to the right, we only need to change 3 cells!

- $w_1 w_2 q w_3 w_4$ – current ID
- $w_1 w_2 x q_1 w_4$ – if $\delta(q, w_3) = (q_1, x, R)$

ϕ_{move}

. . . ensures that C_i yields C_{i+1} by describing legal 2×3 windows of cells. We need 3 cells to cover the cell on the left of the state, the state, and on its right (to enable left and right movements of the head).

$$\phi_{\text{move}} = \bigwedge_{0<i,k<n} \bigvee_{\begin{array}{|c|c|c|}\hline a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 \\ \hline\end{array}\ \text{is legal}} \left(\begin{array}{lll} x_{i,k-1,a_1} & \wedge\, x_{i,k,a_2} & \wedge\, x_{i,k+1,a_3} \wedge \\ x_{i+1,k-1,a_4} & \wedge\, x_{i+1,k,a_5} & \wedge\, x_{i+1,k+1,a_6} \end{array} \right)$$

(Some border cases are not be covered here for simplicity, e.g., i can never be zero.) What is legal depends on the transition function δ.

Example: Let the current ID be $w_1 w_2 q w_3 w_4$ (so, we have blanks before and after it). Whether we go to the left or to the right, we only need to change 3 cells!

- $w_1 w_2 q w_3 w_4$ – current ID
- $w_1 w_2 x q_1 w_4$ – if $\delta(q, w_3) = (q_1, x, R)$
- $w_1 q_2 w_2 y w_4$ – if $\delta(q, w_3) = (q_2, y, L)$

ϕ_{move}

. . . ensures that C_i yields C_{i+1} by describing legal 2×3 windows of cells. We need 3 cells to cover the cell on the left of the state, the state, and on its right (to enable left and right movements of the head).

$$\phi_{\text{move}} = \bigwedge_{0<i,k<n} \bigvee_{\begin{array}{|c|c|c|}\hline a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 \\ \hline\end{array}\ \text{is legal}} \left(\begin{array}{lll} x_{i,k-1,a_1} & \wedge\, x_{i,k,a_2} & \wedge\, x_{i,k+1,a_3} \wedge \\ x_{i+1,k-1,a_4} & \wedge\, x_{i+1,k,a_5} & \wedge\, x_{i+1,k+1,a_6} \end{array} \right)$$

(Some border cases are not be covered here for simplicity, e.g., i can never be zero.)
What is legal depends on the transition function δ.

Example: Let the current ID be $w_1 w_2 q w_3 w_4$ (so, we have blanks before and after it). Whether we go to the left or to the right, we only need to change 3 cells!

- $w_1 w_2 q w_3 w_4$ – current ID
- $w_1 w_2 x q_1 w_4$ – if $\delta(q, w_3) = (q_1, x, R)$
- $w_1 q_2 w_2 y w_4$ – if $\delta(q, w_3) = (q_2, y, L)$

Are we still complete?
We can't seem to be able to move to the left of the initial head position!

ϕ_{move}

. . . ensures that C_i yields C_{i+1} by describing legal 2×3 windows of cells. We need 3 cells to cover the cell on the left of the state, the state, and on its right (to enable left and right movements of the head).

$$\phi_{\text{move}} = \bigwedge_{0<i,k<n} \bigvee_{\begin{array}{|c|c|c|}\hline a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 \\ \hline\end{array}\ \text{is legal}} \begin{pmatrix} x_{i,k-1,a_1} & \wedge\, x_{i,k,a_2} & \wedge\, x_{i,k+1,a_3} \wedge \\ x_{i+1,k-1,a_4} & \wedge\, x_{i+1,k,a_5} & \wedge\, x_{i+1,k+1,a_6} \end{pmatrix}$$

(Some border cases are not be covered here for simplicity, e.g., i can never be zero.)
What is legal depends on the transition function δ.

Example: Let the current ID be $w_1 w_2 q w_3 w_4$ (so, we have blanks before and after it). Whether we go to the left or to the right, we only need to change 3 cells!

- $w_1 w_2 q w_3 w_4$ – current ID
- $w_1 w_2 x q_1 w_4$ – if $\delta(q, w_3) = (q_1, x, R)$
- $w_1 q_2 w_2 y w_4$ – if $\delta(q, w_3) = (q_2, y, L)$

Are we still complete?
We can't seem to be able to move to the left of the initial head position!

- Not a problem: We showed equivalence for semi-infinite tapes under polytime.
- We could alternatively have created a grid of size $(2n)^2$, which also goes n to the left.

ϕ_{accept} – and concluding the Proof

. . . states that the accept state is reached:

$$\phi_{\text{accept}} = \bigvee_{0 \leq i,k \leq n, q_F \in F} x_{i,k,q_F}$$

ϕ_{accept} – and concluding the Proof

. . . states that the accept state is reached:

$$\phi_{\text{accept}} = \bigvee_{0 \leq i,k \leq n, q_F \in F} x_{i,k,q_F}$$

Concluding the Proof:

Recall:

$$\phi_w = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

Finally we check that the size of ϕ_w is polynomial in $|w|$ and that ϕ_w is constructable in polynomial time. (Both is true!)

ϕ_{accept} – and concluding the Proof

. . . states that the accept state is reached:

$$\phi_{\text{accept}} = \bigvee_{0 \leq i,k \leq n, q_F \in F} x_{i,k,q_F}$$

Concluding the Proof:

Recall:

$$\phi_w = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

Finally we check that the size of ϕ_w is polynomial in $|w|$ and that ϕ_w is constructable in polynomial time. (Both is true!)

So, finding a valuation to this formula means deciding $w \in L(M)$ for the arbitrary non-deterministic TM M! So, SAT is **NP**-hard! (It can express every problem in **NP**!)

We have our patient zero now – so, now we can prove **NP**-hardness of other problems by reducing from SAT. (And we build our portfolio...)

NP-complete Problems

CNF-SAT

CNF-SAT is a special case of *SAT*.

$$\textit{CNF-SAT} = \{\ \langle \phi \rangle \mid \phi \text{ is a satisfiable cnf formula } \}$$

where a formula is in cnf (for conjunctive normal form) if it can be generated by a CFG

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

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

Intuitively, a cnf *phi* is simply a conjunction of disjunctions (also called clauses), i.e., $\phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$, where each ϕ_i is a disjunction

CNF-SAT

CNF-SAT is a special case of *SAT*.

$$\textit{CNF-SAT} = \{\ \langle \phi \rangle \mid \phi \text{ is a satisfiable cnf formula } \}$$

where a formula is in cnf (for conjunctive normal form) if it can be generated by a CFG

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

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

Intuitively, a cnf *phi* is simply a conjunction of disjunctions (also called clauses), i.e., $\phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$, where each ϕ_i is a disjunction

Example w8.1

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

CNF-SAT is **NP**-Complete

Clearly *CNF-SAT* is in **NP** because we can again guess an assignment and verify it.

CNF-SAT is **NP**-Complete

Clearly *CNF-SAT* is in **NP** because we can again guess an assignment and verify it.

Giving a **P** reduction from *SAT* to *CNF-SAT* 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. (Note that such an approach could iterate over all lines in a truth table.)

CNF-SAT is **NP**-Complete

Clearly *CNF-SAT* is in **NP** because we can again guess an assignment and verify it.

Giving a **P** reduction from *SAT* to *CNF-SAT* 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. (Note that such an approach could iterate over all lines in a truth table.)

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.

CNF-SAT 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 all two consecutive negations. (This is still satisfaction-preserving.)
2. Translate from nnf to cnf. (This merely preserves satisfiability.)

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 w8.2

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

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 w8.2

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

Pushing Down ¬ cont.

Theorem w8.3

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. □

nnf $\longrightarrow$ cnf

Theorem w8.4

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 $|\phi|$. □

nnf $\longrightarrow$ cnf cont.

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

Example w8.5

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 w8.5

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 w8.5

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:

$$\psi = x_4 \wedge (x_4 \leftrightarrow x_3 \rightarrow x_1) \wedge (x_3 \leftrightarrow x_2 \wedge r) \wedge (x_2 \leftrightarrow p \vee q) \wedge (x_1 \leftrightarrow \neg s)$$

Each conjunct 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 w8.5

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:

$$\psi = x_4 \wedge (x_4 \leftrightarrow x_3 \rightarrow x_1) \wedge (x_3 \leftrightarrow x_2 \wedge r) \wedge (x_2 \leftrightarrow p \vee q) \wedge (x_1 \leftrightarrow \neg s)$$

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

$$\begin{aligned} x_2 \leftrightarrow (p \vee q) &\equiv x_2 \rightarrow (p \vee q) \wedge ((p \vee q) \rightarrow x_2) \\ &\equiv (\neg x_2 \vee p \vee q) \wedge (\neg(p \vee q) \vee x_2) \\ &\equiv (\neg x_2 \vee p \vee q) \wedge ((\neg p \wedge \neg q) \vee x_2) \\ &\equiv (\neg x_2 \vee p \vee q) \wedge (\neg p \vee x_2) \wedge (\neg q \vee x_2) \end{aligned}$$

Conclusion

We proved that *CNF-SAT* is **NP**-hard!

We reduced: $SAT \leq_{\mathbf{P}} nnf \leq_{\mathbf{P}} CNF\text{-}SAT$

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

3-SAT

3-SAT is a special case of *CNF-SAT*.

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

where a formula is in 3cnf (for 3 literal cnf) if it can be generated by a CFG

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

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

3-SAT

3-SAT is a special case of *CNF-SAT*.

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

where a formula is in 3cnf (for 3 literal cnf) if it can be generated by a CFG

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

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

Example w8.6

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

3-SAT is **NP**-Complete

Theorem w8.7

3-SAT is **NP***-Complete*

Proof.

As before: Guess a valuation and verify.

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

Proof: *3-SAT* 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: *3-SAT* 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: *3-SAT* 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: *3-SAT* 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: *3-SAT* 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: *3-SAT* 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: *3-SAT* 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 $u_1, \ldots, u_{k-3}$.

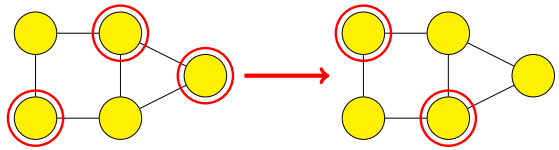
Take $l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5 \vee l_6 \vee l_7$. So $k = 7$ and $k - 3 = 4$. We can write this as:

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

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.)

Vertex Cover vs. Independent Set

Recap from week 7 (more formally)



Theorem w8.8

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

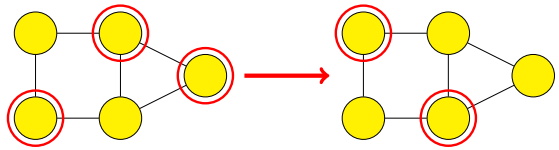
Proof.

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

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

Vertex Cover vs. Independent Set

Recap from week 7 (more formally)



Theorem w8.8

A graph G with $|V| = n$ vertices has a vertex 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.

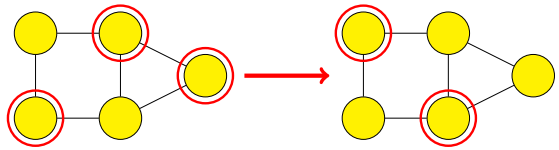
Let G be a graph with n nodes. Let $0 \leq k \leq n$.

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

"$\Rightarrow$" C is a vertex 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 vertex 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.

Vertex Cover vs. Independent Set

Recap from week 7 (more formally)



Theorem w8.8

A graph G with $|V| = n$ vertices has a vertex 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 vertex cover of G iff $V \setminus C$ is an independent set.

"$\Rightarrow$" C is a vertex 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 vertex 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 vertex 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. □

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 verification can also be done in polytime.

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 verification can also be done in polytime.

NP hard Since we saw that both problems can be turned into each other (even trivially), we can choose for which we show hardness! Completeness then follows for both.

We show hardness for Vertex Cover.

Theorem w8.9

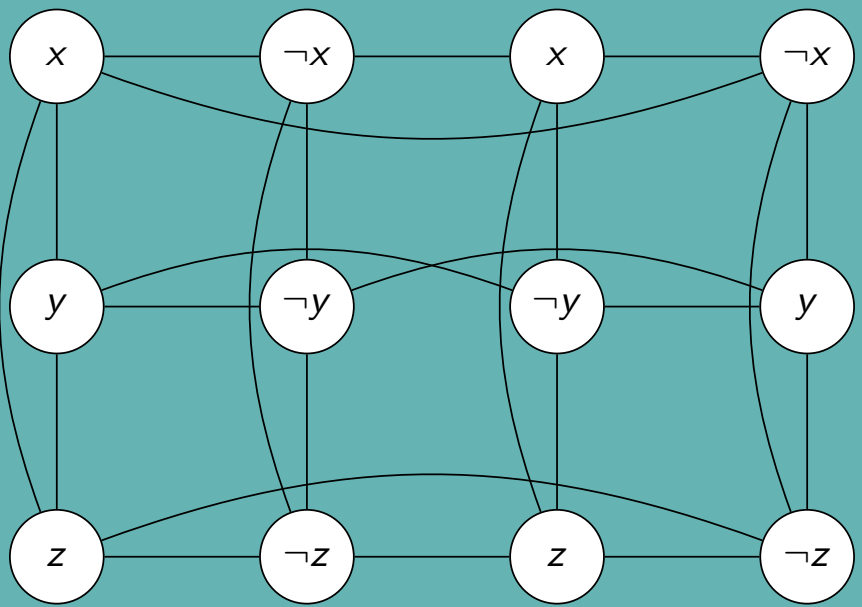
Vertex Cover is **NP***-hard.*

NP-hardness of Vertex Cover

Proof.

We reduce 3-SAT to Vertex Cover.

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



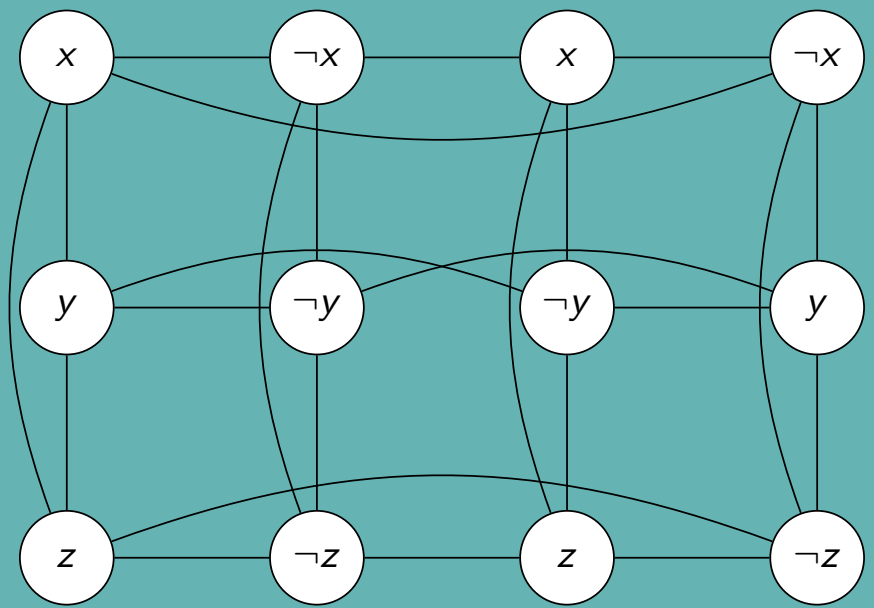
› We have one column per clause.

NP-hardness of Vertex Cover

Proof.

We reduce 3-SAT to Vertex 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.

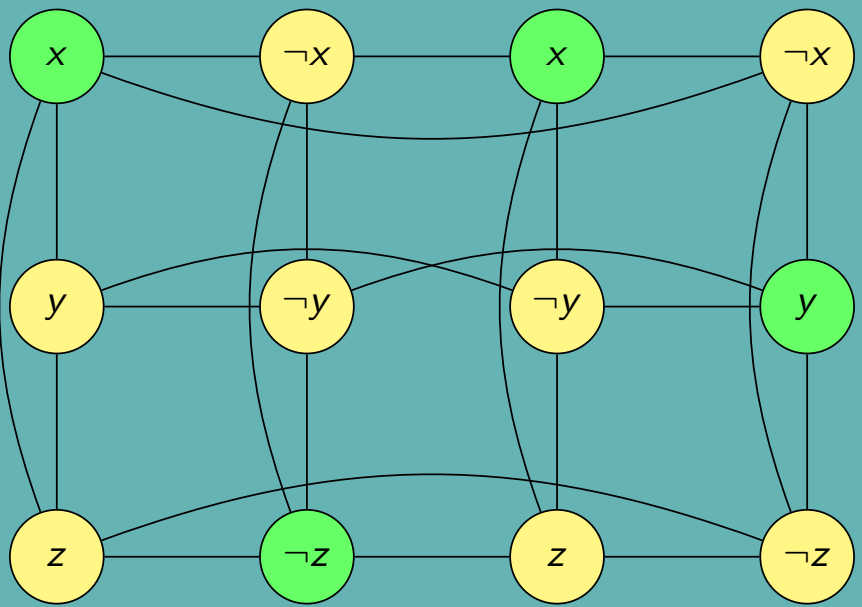
For example, $\pi(x) = \top, \pi(y) = \top, \pi(z) = \bot$ makes the formula true.

NP-hardness of Vertex Cover

Proof.

We reduce 3-SAT to Vertex 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 vertex 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.

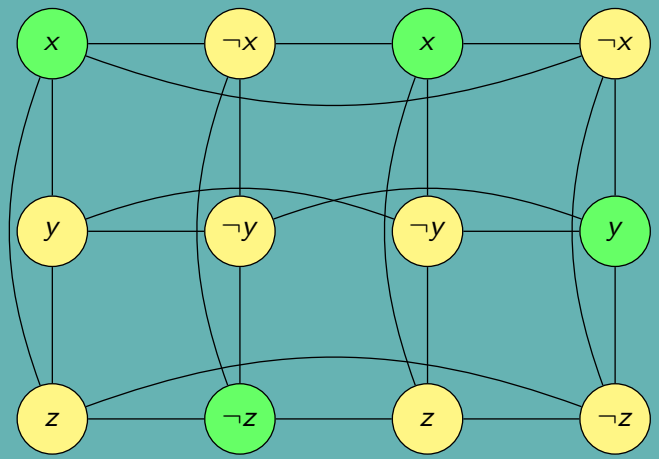
Now we still need to show this claim!

NP-hardness of Vertex Cover (cont'd)

Proof. (Reduction, "⇒").

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

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$, there is (at least) one literal l_i of ϕ_i, s.t. $\phi \models l_i$. E.g., let $l_1, \ldots, l_4$ be the green nodes (non-selected by cover).



NP-hardness of Vertex Cover (cont'd)

Proof. (Reduction, "⇒").

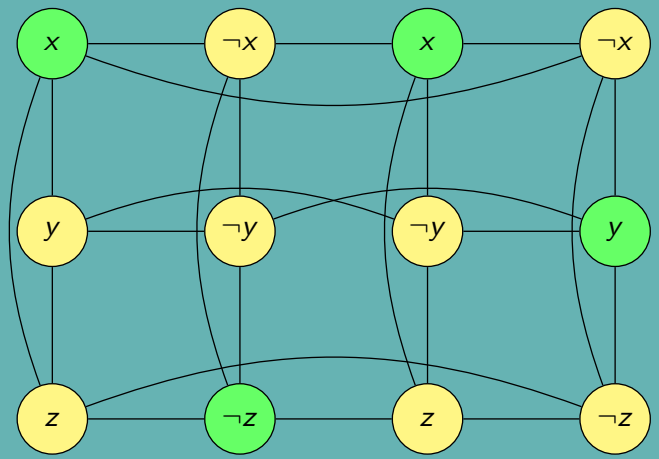
Recall: ϕ is satisfiable iff G has a vertex cover of size $k = 2n$

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$, there is (at least) one literal l_i of ϕ_i, s.t. $\phi \models l_i$. E.g., let $l_1, \ldots, l_4$ be the green nodes (non-selected by cover).

Now define the complement of these nodes as the vertex 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 Vertex Cover (cont'd)

Proof. (Reduction, "⇒").

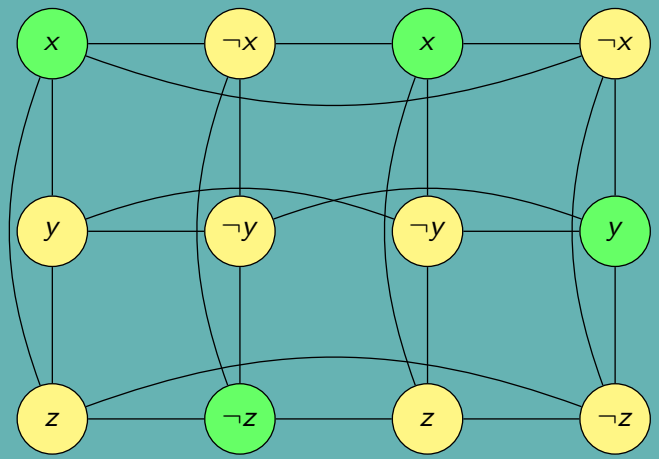
Recall: ϕ is satisfiable iff G has a vertex cover of size $k = 2n$

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$, there is (at least) one literal l_i of ϕ_i, s.t. $\phi \models l_i$. E.g., let $l_1, \ldots, l_4$ be the green nodes (non-selected by cover).

Now define the complement of these nodes as the vertex 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 (yellow) two nodes will *always* cover all edges.

horizontal



NP-hardness of Vertex Cover (cont'd)

Proof. (Reduction, "⇒").

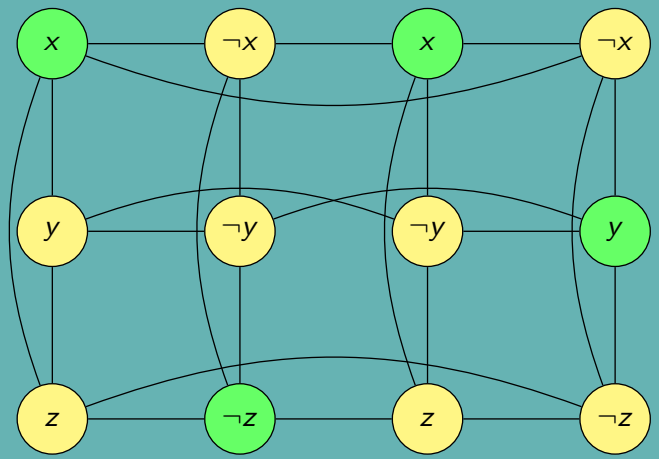
Recall: ϕ is satisfiable iff G has a vertex cover of size $k = 2n$

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$, there is (at least) one literal l_i of ϕ_i, s.t. $\phi \models l_i$. E.g., let $l_1, \ldots, l_4$ be the green nodes (non-selected by cover).

Now define the complement of these nodes as the vertex 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 (yellow) 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 not selected (green), which is impossible since we can't make l_i and $\neg l_i$ true.



NP-hardness of Vertex Cover (cont'd)

Proof. (Reduction, "⇒").

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

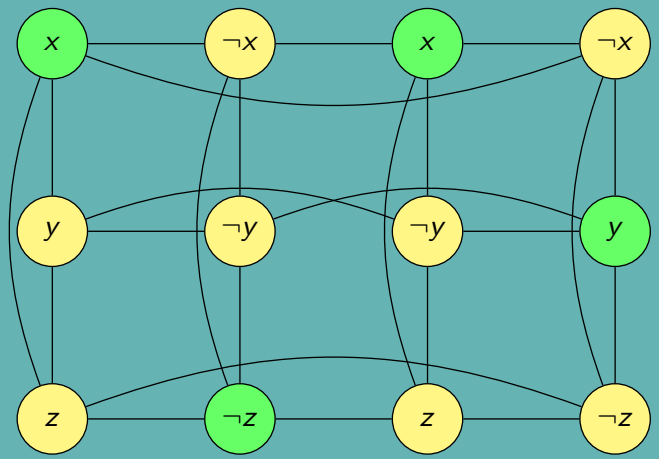
Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$, there is (at least) one literal l_i of ϕ_i, s.t. $\phi \models l_i$. E.g., let $l_1, \ldots, l_4$ be the green nodes (non-selected by cover).

Now define the complement of these nodes as the vertex 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 (yellow) 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 not 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?



NP-hardness of Vertex Cover (cont'd)

Proof. (Reduction, "⇒").

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

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$, there is (at least) one literal l_i of ϕ_i, s.t. $\phi \models l_i$. E.g., let $l_1, \ldots, l_4$ be the green nodes (non-selected by cover).

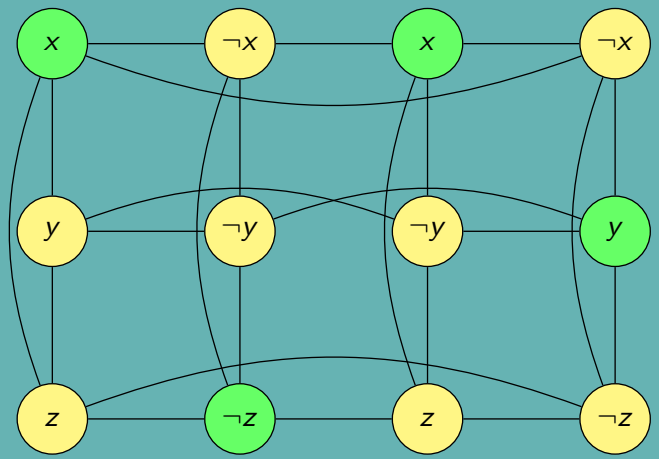
Now define the complement of these nodes as the vertex cover C (the yellow nodes) and show desired properties, i.e., that for each edge (v_1, v_2), at least v_1 or v_2 is in C.

vertical Selecting (yellow) 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 not 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?

A. They forced us to select ≥ 2 literals per clause.



NP-hardness of Vertex Cover (cont'd)

Proof. (Reduction, "⇒").

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

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$, there is (at least) one literal l_i of ϕ_i, s.t. $\phi \models l_i$. E.g., let $l_1, \ldots, l_4$ be the green nodes (non-selected by cover).

Now define the complement of these nodes as the vertex 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.

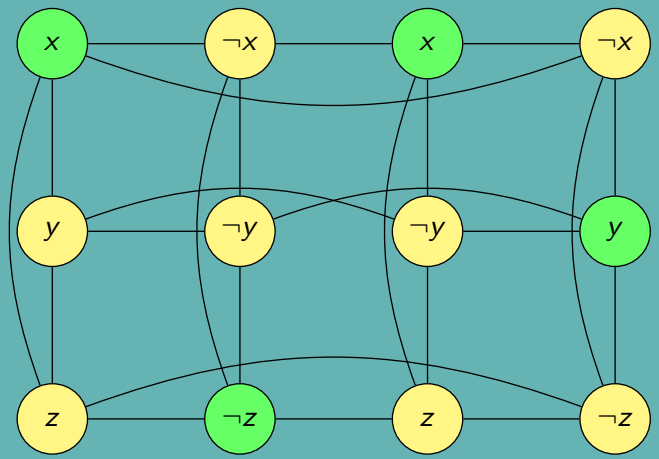
vertical Selecting (yellow) two nodes will *always* cover all edges.

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 not 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?

A. They forced us to select ≥ 2 literals per clause.

Q. What if we select 3 literals?



NP-hardness of Vertex Cover (cont'd)

Proof. (Reduction, "⇒").

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

Let π make ϕ true, $\pi \models \phi$. Then for all clauses $i = 1, ..., n$, there is (at least) one literal l_i of ϕ_i, s.t. $\phi \models l_i$. E.g., let $l_1, \ldots, l_4$ be the green nodes (non-selected by cover).

Now define the complement of these nodes as the vertex 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 (yellow) two nodes will *always* cover all edges.

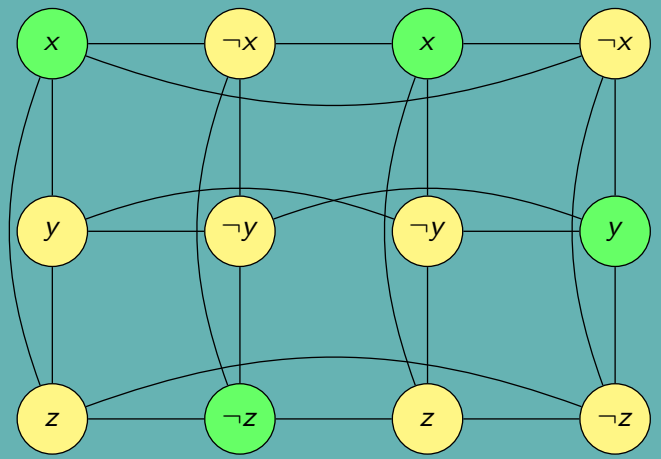
horizontal These edges are always between a variable and its negation. So the only way to *not* have each edge covered is to have both literals not selected (green), which is impossible since we can't make l_i and $\neg l_i$ true.

Q. Why did we need the vertical edges?

A. They forced us to select ≥ 2 literals per clause.

Q. What if we select 3 literals?

A. Then another "column" doesn't work.

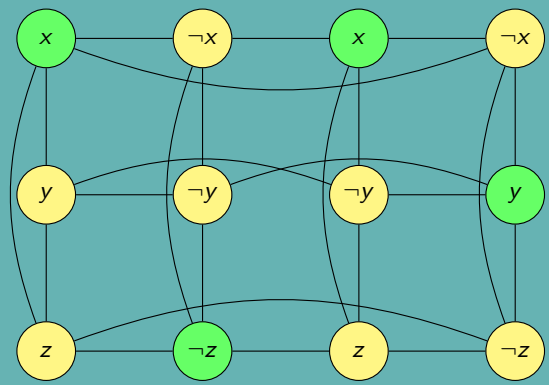


NP-hardness of Vertex Cover (cont'd)

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

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

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



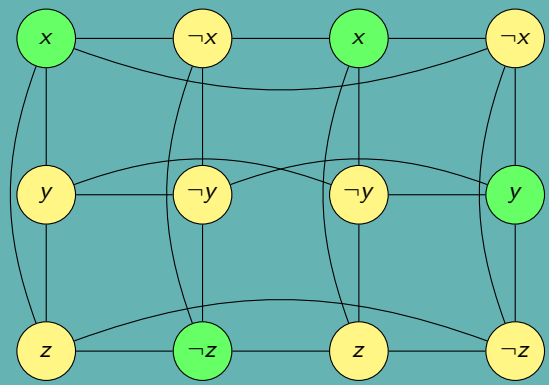
NP-hardness of Vertex Cover (cont'd)

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

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

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

Main argument: Any vertex cover of size $k = 2n$ needs to select precisely 2 elements from each column!



□

NP-hardness of Vertex Cover (cont'd)

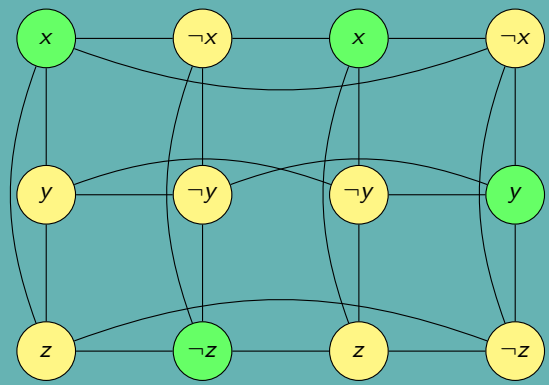
Proof. (Reduction, "$\Leftarrow$").

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

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

Main argument: Any vertex cover of size $k = 2n$ needs to select precisely 2 elements from each column!

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



□

NP-hardness of Vertex Cover (cont'd)

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

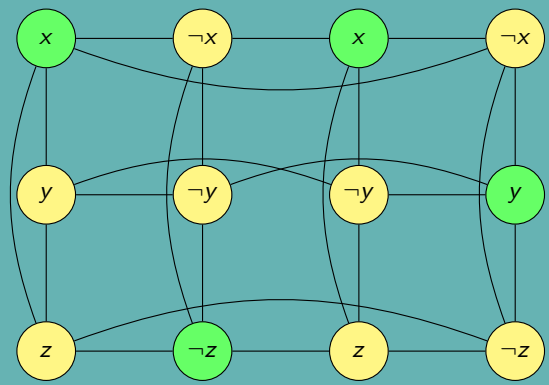
Recall: ϕ is satisfiable iff G has a vertex cover of size $k = 2n$

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

Main argument: Any vertex cover of size $k = 2n$ needs to select precisely 2 elements from each column!

So, why does any vertex 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 Vertex Cover (cont'd)

Proof. (Reduction, "⇐").

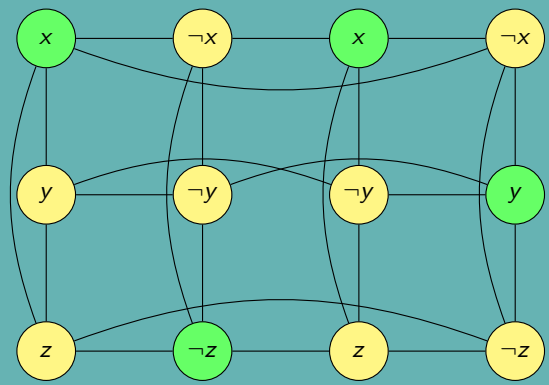
Recall: ϕ is satisfiable iff G has a vertex cover of size $k = 2n$

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

Main argument: Any vertex cover of size $k = 2n$ needs to select precisely 2 elements from each column!

So, why does any vertex 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?

□

NP-hardness of Vertex Cover (cont'd)

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

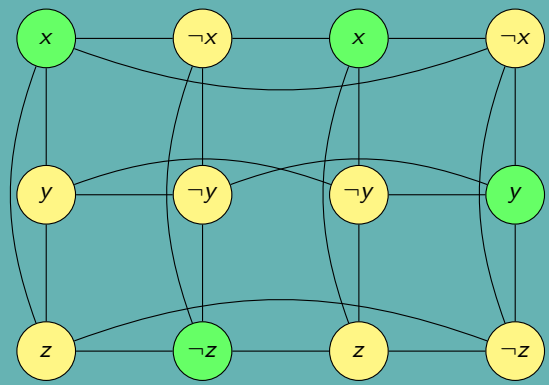
Recall: ϕ is satisfiable iff G has a vertex cover of size $k = 2n$

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

Main argument: Any vertex cover of size $k = 2n$ needs to select precisely 2 elements from each column!

So, why does any vertex 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, $\pi(l_i) = \pi(\neg l_i) = \top$.
- Can't happen! They all share a (horizontal) edge, so using both in the assignment (green) would exclude them for the vertex cover – leaving a non-covered edge.

□

Summary of **NP**-complete Problems

We proved the **NP**-completeness of:

- SAT, CNF-SAT, 3-SAT
- Set Cover, Independent Set (from Karp's 21)

In the tutorials, you will see:

- k-SAT
- Partition-Clique
- Graph Colouring, 3-Colouring (from Karp's 21)
- Integer Linear Programs (ILPs)

Other important **NP**-complete problems you should know:

- Karp's 21 **NP**-complete problems (from his 1972 paper)
- Clique (from Karp's 21)
- Hamiltonian Path (a variant from one of Kar's 21)
- Travelling Salesman Problem (from 1930 or earlier)

co-NP and **co-P**

The class co-NP

Definition w8.1

A problem is in **co-X** if and only if its complement is in **X**.

Phrased more formally, applied to $\mathbf{X} = \mathbf{NP}$: Let $\bar{L} = \Sigma^* \setminus L$

$$\textbf{co-NTIME}(t(n)) = \{L \mid \bar{L} \in \textbf{NTIME}(t(n))\} \qquad \textbf{co-NP} = \bigcup_{k \in \mathbb{N}} \textbf{co-NTIME}(n^k)$$

The class co-NP

Definition w8.1

A problem is in **co-X** if and only if its complement is in **X**.

Phrased more formally, applied to $\mathbf{X} = \mathbf{NP}$: Let $\bar{L} = \Sigma^* \setminus L$

$$\textbf{co-NTIME}(t(n)) = \{L \mid \bar{L} \in \textbf{NTIME}(t(n))\} \qquad \textbf{co-NP} = \bigcup_{k \in \mathbb{N}} \textbf{co-NTIME}(n^k)$$

Key Messages (applied to $\mathbf{X} = \mathbf{NP}$)

- No matter whether $w \in L$ or $w \notin L$, the decision can be made in (non-deterministic) polytime, both for $L \in \mathbf{NP}$ and/or $L \in \mathbf{co\text{-}NP}$.
- For **NP** versus **co-NP** problems, we get different properties:

The class co-NP

Definition w8.1

A problem is in **co-X** if and only if its complement is in **X**.

Phrased more formally, applied to $\mathbf{X} = \mathbf{NP}$: Let $\bar{L} = \Sigma^* \setminus L$

$$\textbf{co-NTIME}(t(n)) = \{L \mid \bar{L} \in \textbf{NTIME}(t(n))\} \qquad \textbf{co-NP} = \bigcup_{k \in \mathbb{N}} \textbf{co-NTIME}(n^k)$$

Key Messages (applied to $\mathbf{X} = \mathbf{NP}$)

- No matter whether $w \in L$ or $w \notin L$, the decision can be made in (non-deterministic) polytime, both for $L \in \mathbf{NP}$ and/or $L \in \mathbf{co\text{-}NP}$.
- For **NP** versus **co-NP** problems, we get different properties:
  - For $L \in \mathbf{NP}$, we can provide a certificate for yes-instances (i.e., $w \in L$)
    - $w \in L$: The non-det. TM (for L) accepts w on at least one path.
    - $w \notin L$: The non-det. TM (for L) rejects w on all paths.
  - For $L \in \mathbf{co\text{-}NP}$, we can provide a certificate for no-instances (i.e., $w \notin L$)
    - $w \in L$: The non-det. TM (for $\bar{L}$) reject w on all paths.
    - $w \notin L$: The non-det. TM (for $\bar{L}$) accepts w on at least one path.

Properties of **co-NP** and **co-P**

Note that it's still not known whether $\mathbf{NP} \overset{?}{=} \mathbf{co\text{-}NP}$.

Properties of co-NP and co-P

Note that it's still not known whether $\mathbf{NP} \stackrel{?}{=} \mathbf{co\text{-}NP}$.

Theorem w8.2

1. $\mathbf{P} \subseteq \mathbf{co\text{-}NP}$ *(thus:* $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co\text{-}NP}$*)*
2. *If* $\mathbf{P} = \mathbf{NP}$*, then* $\mathbf{P} = \mathbf{NP} = \mathbf{co\text{-}NP}$*.*

Properties of co-NP and co-P

Note that it's still not known whether $\mathbf{NP} \stackrel{?}{=} \mathbf{co\text{-}NP}$.

Theorem w8.2

1. $\mathbf{P} \subseteq \mathbf{co\text{-}NP}$ *(thus:* $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co\text{-}NP}$*)*
2. *If* $\mathbf{P} = \mathbf{NP}$, *then* $\mathbf{P} = \mathbf{NP} = \mathbf{co\text{-}NP}$.

Proof.

Because **P** is closed under complementation. □

Properties of co-NP and co-P

Note that it's still not known whether $\mathbf{NP} \overset{?}{=} \mathbf{co\text{-}NP}$.

Theorem w8.2

1. $\mathbf{P} \subseteq \mathbf{co\text{-}NP}$ *(thus:* $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co\text{-}NP}$*)*
2. *If* $\mathbf{P} = \mathbf{NP}$*, then* $\mathbf{P} = \mathbf{NP} = \mathbf{co\text{-}NP}$*.*

Proof.

Because **P** is closed under complementation. □

Once more on **NP** vs. **co-NP**

- For $L \in \mathbf{NP}$ we have a (poly-size) certificate for YES-instances (i.e., $w \in L$)
- For $L \in \mathbf{co\text{-}NP}$ we have a (poly-size) certificate for NO-instances (i.e., $w \notin L$)
- What if $\mathbf{NP} = \mathbf{co\text{-}NP}$?

Properties of co-NP and co-P

Note that it's still not known whether $\mathbf{NP} \stackrel{?}{=} \mathbf{co\text{-}NP}$.

Theorem w8.2

1. $\mathbf{P} \subseteq \mathbf{co\text{-}NP}$ *(thus:* $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co\text{-}NP}$*)*
2. *If* $\mathbf{P} = \mathbf{NP}$*, then* $\mathbf{P} = \mathbf{NP} = \mathbf{co\text{-}NP}$*.*

Proof.

Because **P** is closed under complementation. □

Once more on **NP** vs. **co-NP**

- For $L \in \mathbf{NP}$ we have a (poly-size) certificate for YES-instances (i.e., $w \in L$)
- For $L \in \mathbf{co\text{-}NP}$ we have a (poly-size) certificate for NO-instances (i.e., $w \notin L$)
- What if $\mathbf{NP} = \mathbf{co\text{-}NP}$? We have (poly-size) certificate for any instance.
- What if $\mathbf{NP} \neq \mathbf{co\text{-}NP}$?

Properties of co-NP and co-P

Note that it's still not known whether $\mathbf{NP} \stackrel{?}{=} \mathbf{co\text{-}NP}$.

Theorem w8.2

1. $\mathbf{P} \subseteq \mathbf{co\text{-}NP}$ *(thus:* $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co\text{-}NP}$*)*
2. *If* $\mathbf{P} = \mathbf{NP}$*, then* $\mathbf{P} = \mathbf{NP} = \mathbf{co\text{-}NP}$*.*

Proof.

Because **P** is closed under complementation. □

Once more on **NP** vs. **co-NP**

- For $L \in \mathbf{NP}$ we have a (poly-size) certificate for YES-instances (i.e., $w \in L$)
- For $L \in \mathbf{co\text{-}NP}$ we have a (poly-size) certificate for NO-instances (i.e., $w \notin L$)
- What if $\mathbf{NP} = \mathbf{co\text{-}NP}$? We have (poly-size) certificate for any instance.
- What if $\mathbf{NP} \neq \mathbf{co\text{-}NP}$? Then, there are problems that guarantee poly-size certificates for only one answer.

co-NP-Hardness and -Completeness vs. NP ...

Remark: Hardness and Completeness for **Co-NP** are defined as for any other class:

Definition w8.3

A problem B is **co-NP**-hard if every $A \in$ **co-NP** is **P**-reducible to B.
A problem B is **co-NP**-complete if it's in **co-NP** and **co-NP**-hard.

co-NP-Hardness and -Completeness vs. NP ...

Remark: Hardness and Completeness for **Co-NP** are defined as for any other class:

Definition w8.3

A problem B is **co-NP**-hard if every $A \in$ **co-NP** is **P**-reducible to B.
A problem B is **co-NP**-complete if it's in **co-NP** and **co-NP**-hard.

So, what problems are "harder" **NP** or **co-NP**? I.e., which problem would you rather attempt to solve? One that's **NP**-complete or one that's **co-NP**-complete?

co-NP-Hardness and -Completeness vs. NP ...

Remark: Hardness and Completeness for **Co-NP** are defined as for any other class:

Definition w8.3

A problem B is **co-NP**-hard if every $A \in$ **co-NP** is **P**-reducible to B.
A problem B is **co-NP**-complete if it's in **co-NP** and **co-NP**-hard.

So, what problems are "harder" **NP** or **co-NP**? I.e., which problem would you rather attempt to solve? One that's **NP**-complete or one that's **co-NP**-complete?

That depends on our technology and assumptions!

- Solvers are usually optimized to find solutions, not to disprove their existence.
- Thus, if you expect your problem to have the respective property, **NP** works better, otherwise **co-NP**. Thus,
  - Proving $w \in L$ for L **NP**-complete is usually easy, proving $w \notin L$ is hard.
  - Proving $w \notin L$ for L **co-NP**-complete is usually easy, proving $w \in L$ is hard.

  Because solvers find certificates!

Variants of SAT

Let's go through SAT:

- SAT =

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $=$

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in \mathit{SAT}$ iff $\langle\phi\rangle \notin \mathit{UNSAT}$)

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in \mathit{SAT}$ iff $\langle\phi\rangle \notin \mathit{UNSAT}$)
- TAUT $=$

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in \mathit{SAT}$ iff $\langle\phi\rangle \notin \mathit{UNSAT}$)
- TAUT $= \{\langle\phi\rangle \mid \phi$ has no unsatisfying valuations $=$ has only satisfying valuations$\}$

How do they relate?

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in \mathit{SAT}$ iff $\langle\phi\rangle \notin \mathit{UNSAT}$)
- TAUT $= \{\langle\phi\rangle \mid \phi$ has no unsatisfying valuations $=$ has only satisfying valuations$\}$

How do they relate?

- UNSAT:
  - $\phi \in \mathit{UNSAT}$: all valuations make ϕ wrong
  - $\phi \notin \mathit{UNSAT}$: there is a valuation that makes ϕ true

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in \mathit{SAT}$ iff $\langle\phi\rangle \notin \mathit{UNSAT}$)
- TAUT $= \{\langle\phi\rangle \mid \phi$ has no unsatisfying valuations $=$ has only satisfying valuations$\}$

How do they relate?

- UNSAT: Looks like co-NP...
  - $\phi \in \mathit{UNSAT}$: all valuations make ϕ wrong
  - $\phi \notin \mathit{UNSAT}$: there is a valuation that makes ϕ true

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in SAT$ iff $\langle\phi\rangle \notin UNSAT$)
- TAUT $= \{\langle\phi\rangle \mid \phi$ has no unsatisfying valuations $=$ has only satisfying valuations$\}$

How do they relate?

- UNSAT: Looks like co-NP...
  - $\phi \in UNSAT$: all valuations make ϕ wrong
  - $\phi \notin UNSAT$: there is a valuation that makes ϕ true
- TAUT:
  - $\phi \in TAUT$: all valuations make ϕ true
  - $\phi \notin TAUT$: there is a valuation that makes ϕ false

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in SAT$ iff $\langle\phi\rangle \notin UNSAT$)
- TAUT $= \{\langle\phi\rangle \mid \phi$ has no unsatisfying valuations $=$ has only satisfying valuations$\}$

How do they relate?

- UNSAT: Looks like co-NP...
  - $\phi \in UNSAT$: all valuations make ϕ wrong
  - $\phi \notin UNSAT$: there is a valuation that makes ϕ true
- TAUT: Looks again like co-NP...
  - $\phi \in TAUT$: all valuations make ϕ true
  - $\phi \notin TAUT$: there is a valuation that makes ϕ false

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in SAT$ iff $\langle\phi\rangle \notin UNSAT$)
- TAUT $= \{\langle\phi\rangle \mid \phi$ has no unsatisfying valuations $=$ has only satisfying valuations$\}$

How do they relate?

- UNSAT: Looks like co-NP...
  - $\phi \in UNSAT$: all valuations make ϕ wrong
  - $\phi \notin UNSAT$: there is a valuation that makes ϕ true
- TAUT: Looks again like co-NP...
  - $\phi \in TAUT$: all valuations make ϕ true
  - $\phi \notin TAUT$: there is a valuation that makes ϕ false

Does a negation help (or not)?

- $\phi \in SAT$ iff $\neg\phi$

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in SAT$ iff $\langle\phi\rangle \notin UNSAT$)
- TAUT $= \{\langle\phi\rangle \mid \phi$ has no unsatisfying valuations $=$ has only satisfying valuations$\}$

How do they relate?

- UNSAT: Looks like co-NP...
  - $\phi \in UNSAT$: all valuations make ϕ wrong
  - $\phi \notin UNSAT$: there is a valuation that makes ϕ true
- TAUT: Looks again like co-NP...
  - $\phi \in TAUT$: all valuations make ϕ true
  - $\phi \notin TAUT$: there is a valuation that makes ϕ false

Does a negation help (or not)?

- $\phi \in SAT$ iff $\neg\phi \notin TAUT$ (So you see "it's flipped")
- $\phi \in TAUT$ iff $\neg\phi$

Variants of SAT

Let's go through SAT:

- SAT $= \{\langle\phi\rangle \mid \phi$ has a satisfying valuation$\}$
- UNSAT $= \{\langle\phi\rangle \mid \phi$ has no satisfying valuation$\}$ (Note $\langle\phi\rangle \in SAT$ iff $\langle\phi\rangle \notin UNSAT$)
- TAUT $= \{\langle\phi\rangle \mid \phi$ has no unsatisfying valuations $=$ has only satisfying valuations$\}$

How do they relate?

- UNSAT: Looks like co-NP...
  - $\phi \in UNSAT$: all valuations make ϕ wrong
  - $\phi \notin UNSAT$: there is a valuation that makes ϕ true
- TAUT: Looks again like co-NP...
  - $\phi \in TAUT$: all valuations make ϕ true
  - $\phi \notin TAUT$: there is a valuation that makes ϕ false

Does a negation help (or not)?

- $\phi \in SAT$ iff $\neg\phi \notin TAUT$ (So you see "it's flipped")
- $\phi \in TAUT$ iff $\neg\phi \in UNSAT$ (No "flipping" here)

TAUT is **co-NP**-complete.

Theorem w8.4

TAUT is **co-NP***-complete.*

Proof.

- Hardness: Covered in the tutorial.
- Membership:
  - Guess a valuation π.
  - Return true iff π makes the formula false.

  Correct?

□

TAUT is **co-NP**-complete.

Theorem w8.4

TAUT is **co-NP***-complete.*

Proof.

- Hardness: Covered in the tutorial.
- Membership:
  - Guess a valuation π.
  - Return true iff π makes the formula false.

  Correct? No, that is an NP proof, and a wrong one: Clearly, if we can make it false, it's not a tautology.

□

TAUT is **co-NP**-complete.

Theorem w8.4

TAUT is **co-NP***-complete.*

Proof.

- Hardness: Covered in the tutorial.
- Membership:
  - Guess a valuation π.
  - Return true iff π makes the formula true.

  Now?

□

TAUT is **co-NP**-complete.

Theorem w8.4

TAUT is **co-NP***-complete.*

Proof.

- Hardness: Covered in the tutorial.
- Membership:
  - Guess a valuation π.
  - Return true iff π makes the formula true.

  Now? No, that's still an NP proof (we guess to say YES!), and a wrong one again: It's not sufficient to have one true assignment, all must be true!

□

TAUT is **co-NP**-complete.

Theorem w8.4

TAUT is **co-NP***-complete.*

Proof.

- Hardness: Covered in the tutorial.
- Membership:
  - Guess a valuation π.
  - Return false iff π makes the formula true.

Now?!

□

TAUT is **co-NP**-complete.

Theorem w8.4

TAUT is **co-NP***-complete.*

Proof.

- Hardness: Covered in the tutorial.
- Membership:
  - Guess a valuation π.
  - Return false iff π makes the formula true.

  Now?! Finally, we return false after the guessing, but ... this really doesn't make sense: there's nothing wrong with finding a true assignment...

□

TAUT is **co-NP**-complete.

Theorem w8.4

TAUT is **co-NP***-complete.*

Proof.

- Hardness: Covered in the tutorial.
- Membership:
  - Guess a valuation π.
  - Return false iff π makes the formula false.

  Now! :) We return false if we can guess an assignment that makes it false. (And true otherwise.)

□

Keep in mind: For **co-NP** membership we need to show that the complement is in **NP**. The complement of a tautology is that that there exists an assignment that makes it true. That's what checked in **NP**.

On the Hardness of *TAUT*

Theorem w8.5

If TAUT is in **P**, *then every problem in* **NP** *is in* **P**.

Proof.

We show that we could solve any *SAT* problem in **P** if *TAUT* is in **P**. (*SAT* is **NP**-hard!)

On the Hardness of *TAUT*

Theorem w8.5

If TAUT is in **P**, *then every problem in* **NP** *is in* **P**.

Proof.

We show that we could solve any *SAT* problem in **P** if *TAUT* is in **P**. (*SAT* is **NP**-hard!)

- A formula ϕ is satisfiable if $\neg\phi$ is not a tautology. (You can easily prove this.)
  E.g., $\phi = (x \vee \neg y) \wedge y$, $\neg\phi = (\neg x \wedge y) \vee \neg y$.
  For $\pi(x) = \top$ and $\pi(y) = \top$ we get $\pi \models \phi$ and $\pi \not\models \neg\phi$.
- Solve *SAT* in polytime:
  - If ϕ is the input, run *TAUT* on $\neg\phi$.
  - flip the result.

□

On the Hardness of *TAUT*

Theorem w8.5

If TAUT is in **P**, *then every problem in* **NP** *is in* **P**.

Proof.

We show that we could solve any *SAT* problem in **P** if *TAUT* is in **P**. (*SAT* is **NP**-hard!)

- A formula ϕ is satisfiable if $\neg\phi$ is not a tautology. (You can easily prove this.)
  E.g., $\phi = (x \vee \neg y) \wedge y$, $\neg\phi = (\neg x \wedge y) \vee \neg y$.
  For $\pi(x) = \top$ and $\pi(y) = \top$ we get $\pi \models \phi$ and $\pi \not\models \neg\phi$.
- Solve *SAT* in polytime:
  - If ϕ is the input, run *TAUT* on $\neg\phi$.
  - flip the result.

□

Question

- Have we shown that *TAUT* is **NP**-hard?

On the Hardness of *TAUT*

Theorem w8.5

If TAUT is in **P**, *then every problem in* **NP** *is in* **P**.

Proof.

We show that we could solve any *SAT* problem in **P** if *TAUT* is in **P**. (*SAT* is **NP**-hard!)

- A formula ϕ is satisfiable if $\neg\phi$ is not a tautology. (You can easily prove this.)
  E.g., $\phi = (x \vee \neg y) \wedge y$, $\neg\phi = (\neg x \wedge y) \vee \neg y$.
  For $\pi(x) = \top$ and $\pi(y) = \top$ we get $\pi \models \phi$ and $\pi \not\models \neg\phi$.
- Solve *SAT* in polytime:
  - If ϕ is the input, run *TAUT* on $\neg\phi$.
  - flip the result.

□

Questions

- Have we shown that *TAUT* is **NP**-hard?
- **No!** This was not a polytime reduction from *SAT* to *TAUT*. Why?

On the Hardness of *TAUT*

Theorem w8.5

If TAUT is in **P**, *then every problem in* **NP** *is in* **P**.

Proof.

We show that we could solve any *SAT* problem in **P** if *TAUT* is in **P**. (*SAT* is **NP**-hard!)

- A formula ϕ is satisfiable if $\neg\phi$ is not a tautology. (You can easily prove this.)
  E.g., $\phi = (x \vee \neg y) \wedge y$, $\neg\phi = (\neg x \wedge y) \vee \neg y$.
  For $\pi(x) = \top$ and $\pi(y) = \top$ we get $\pi \models \phi$ and $\pi \not\models \neg\phi$.
- Solve *SAT* in polytime:
  - If ϕ is the input, run *TAUT* on $\neg\phi$.
  - flip the result.

□

Questions

- Have we shown that *TAUT* is **NP**-hard?
- **No!** This was not a polytime reduction from *SAT* to *TAUT*. Why?
- Because we flipped the result! We don't implement $w \in SAT$ iff $f(w) \in TAUT$.