Satisfiability

Recall Horn formulas are easy to solve

Satisfiability

Recall Horn formulas are easy to solve

How about more general formulas: CNF (conjunction normal form)?

Satisfiability

Recall Horn formulas are easy to solve

How about more general formulas: CNF (conjunction normal form)?

Definition

AND

A **CNF formula** is a $\boxed{\text{conjunction}}$ of clauses, where each clause is a disjunction of literals

OR

Satisfiability

Recall Horn formulas are easy to solve

How about more general formulas: CNF (conjunction normal form)?

Definition

A **CNF formula** is a conjunction of clauses, where each clause is a disjunction of literals

Example: $(\overset{x_1}{\cancel{x_1}} \vee x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge (\bar{x}_4 \vee x_7)$

Satisfiability

Recall Horn formulas are easy to solve

How about more general formulas: CNF (conjunction normal form)?

Definition

A **CNF formula** is a conjunction of clauses, where each clause is a disjunction of literals

Example: $(v_1 \vee x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge (\bar{x}_4 \vee x_7)$

Definition

A **k-CNF** is a CNF where each clause contains exactly k literals

The Satisfiability Problem (SAT)

The Satisfiability Problem (SAT)

Instance: A CNF Φ

The Satisfiability Problem (SAT)

Instance: A CNF Φ

Objective: Decide if Φ is satisfiable, i.e., is there an assignment so that Φ is true?

The Satisfiability Problem

The Satisfiability Problem (SAT)

Instance: A CNF Φ

Objective: Decide if Φ is satisfiable, i.e., is there an assignment so that Φ is true?

The k-Satisfiability Problem (k-SAT)

Instance: A k-CNF Φ

Objective: Decide if Φ is satisfiable

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

Proof.

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

Proof. First consider an intuition for solving SAT:

$$\Phi = (x_1 \lor x_2 \lor x_3) \land (x_2 \lor \bar{x}_3 \lor x_4) \land (x_3 \lor \bar{x}_1 \lor x_5)$$

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

Proof. First consider an intuition for solving SAT:

- pick one literal from each clause

$$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$$

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

Proof. First consider an intuition for solving SAT:

- pick one literal from each clause
- select an assignment that satisfies all selected literals

$$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$$

$x_1 = 1$, $x_2 = 1$, $x_3 = 1$

x_4, x_5

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

Proof. First consider an intuition for solving SAT:

- pick one literal from each clause
- select an assignment that satisfies all selected literals
- make sure there's no conflict:

$$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$$

$x_3 = 0$ $x_3 = 1$

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

Proof. First consider an intuition for solving SAT:

- pick one literal from each clause
- select an assignment that satisfies all selected literals
- make sure there's no conflict: Don't pick x from one clause and $\bar{x}$ from another

$$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$$

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

Proof. First consider an intuition for solving SAT:

- pick one literal from each clause
- select an assignment that satisfies all selected literals
- make sure there's no conflict: Don't pick x from one clause and $\bar{x}$ from another

$$\Phi = (\underset{\substack{\uparrow \\ \text{good}}}{x_1} \vee x_2 \vee x_3) \wedge (\underset{\uparrow}{x_2} \vee \bar{x}_3 \vee x_4) \wedge (\underset{\uparrow}{x_3} \vee \bar{x}_1 \vee x_5)$$

3-SAT and Independent Set

Theorem

3-SAT $\leq_P$ Independent Set

Proof. First consider an intuition for solving SAT:

- pick one literal from each clause
- select an assignment that satisfies all selected literals
- make sure there's no conflict: Don't pick x from one clause and $\bar{x}$ from another

$$\Phi = (\overset{\overset{\text{bad}}{\downarrow}}{\underset{\underset{\text{good}}{\uparrow}}{x_1}} \vee x_2 \vee x_3) \wedge (\underset{\uparrow}{x_2} \vee \overset{\downarrow}{\bar{x}_3} \vee x_4) \wedge (\overset{\downarrow}{\underset{\uparrow}{x_3}} \vee \bar{x}_1 \vee x_5)$$

Theorem

3-SAT $\leq_P$ Independent Set

Proof. First consider an intuition for solving SAT:

- pick one literal from each clause
- select an assignment that satisfies all selected literals
- make sure there's no conflict: Don't pick x from one clause and $\bar{x}$ from another

$$\Phi = (\overset{\substack{\text{bad}\\ \downarrow}}{\underset{\substack{\uparrow\\ \text{good}}}{x_1}} \vee x_2 \vee x_3) \wedge (\underset{\uparrow}{x_2} \vee \overset{\downarrow}{\bar{x}_3} \vee x_4) \wedge (\overset{\downarrow}{\underset{\uparrow}{x_3}} \vee \bar{x}_1 \vee x_5)$$

We encode a CNF as a graph, and encode an assignment as independent sets (to keep track of the conflicts)

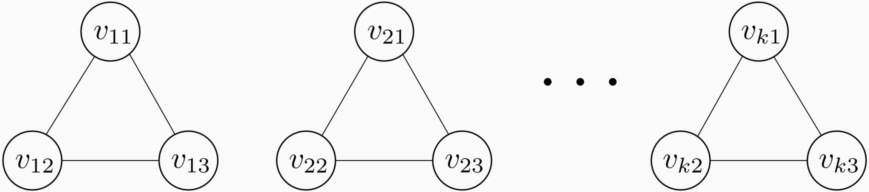
Consider a 3-SAT instance with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_k$

Consider a 3-SAT instance with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_k$

We build a graphs $G = (V, E)$ with $3k$ vertices,

Consider a 3-SAT instance with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_k$

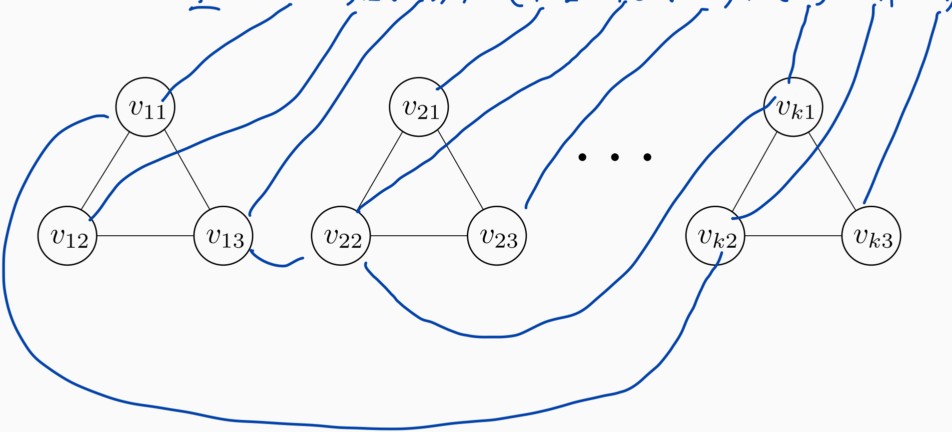
We build a graphs $G = (V, E)$ with $3k$ vertices, grouped into k triangles.



Consider a 3-SAT instance with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_k$

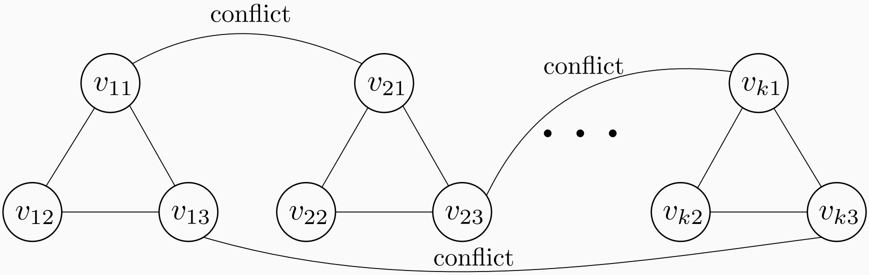
We build a graphs $G = (V, E)$ with $3k$ vertices, grouped into k triangles. Each triangle contains v_{i1}, v_{i2}, v_{i3} where v_{ij} corresponds to the j-th literal in C_i.

$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$



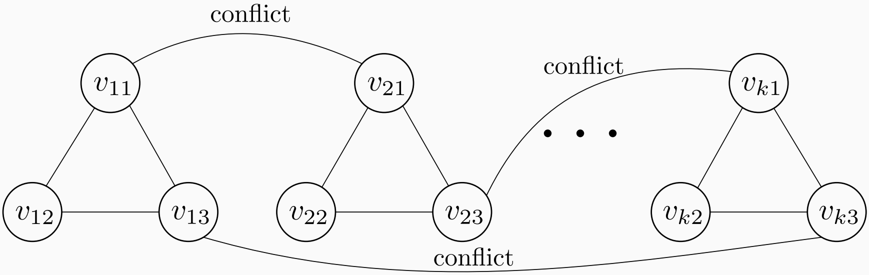
Consider a 3-SAT instance with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_k$

We build a graphs $G = (V, E)$ with $3k$ vertices, grouped into k triangles. Each triangle contains v_{i1}, v_{i2}, v_{i3} where v_{ij} corresponds to the j-th literal in C_i. Add edges for conflicts, i.e., x_j and $\bar{x}_j$:



Consider a 3-SAT instance with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_k$

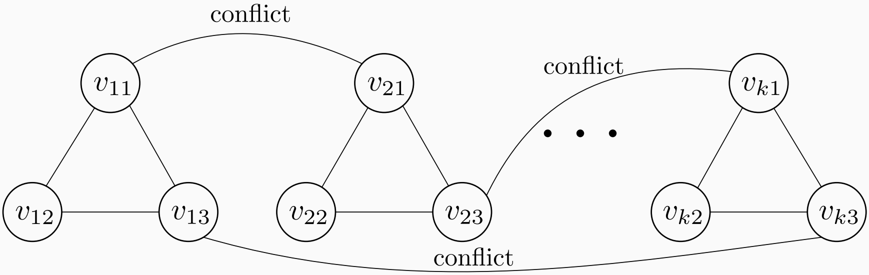
We build a graphs $G = (V, E)$ with $3k$ vertices, grouped into k triangles. Each triangle contains v_{i1}, v_{i2}, v_{i3} where v_{ij} corresponds to the j-th literal in C_i. Add edges for conflicts, i.e., x_j and $\bar{x}_j$:



At most one vertex in each triangle can be in an independent set,

Consider a 3-SAT instance with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_k$

We build a graphs $G = (V, E)$ with $3k$ vertices, grouped into k triangles. Each triangle contains v_{i1}, v_{i2}, v_{i3} where v_{ij} corresponds to the j-th literal in C_i. Add edges for conflicts, i.e., x_j and $\bar{x}_j$:



At most one vertex in each triangle can be in an independent set, so the size of an independent set cannot be larger than k

- If there exists a satisfying assignment, there exists a satisfied literal in each clause (triangle).

- If there exists a satisfying assignment, there exists a satisfied literal in each clause (triangle). Pick such a literal and include it into the independent set

- If there exists a satisfying assignment, there exists a satisfied literal in each clause (triangle). Pick such a literal and include it into the independent set

  There is no conflicts. It's in fact an independent set

- If there exists a satisfying assignment, there exists a satisfied literal in each clause (triangle). Pick such a literal and include it into the independent set
  There is no conflicts. It's in fact an independent set
- If there exists an independent set S of size k, every triangle contains a vertex from S.

- If there exists a satisfying assignment, there exists a satisfied literal in each clause (triangle). Pick such a literal and include it into the independent set
  There is no conflicts. It's in fact an independent set
- If there exists an independent set S of size k, every triangle contains a vertex from S. We can choose an assignment so that all literals (vertices of S) are satisfied — there's no conflicts

- If there exists a satisfying assignment, there exists a satisfied literal in each clause (triangle). Pick such a literal and include it into the independent set

  There is no conflicts. It's in fact an independent set
- If there exists an independent set S of size k, every triangle contains a vertex from S. We can choose an assignment so that all literals (vertices of S) are satisfied — there's no conflicts

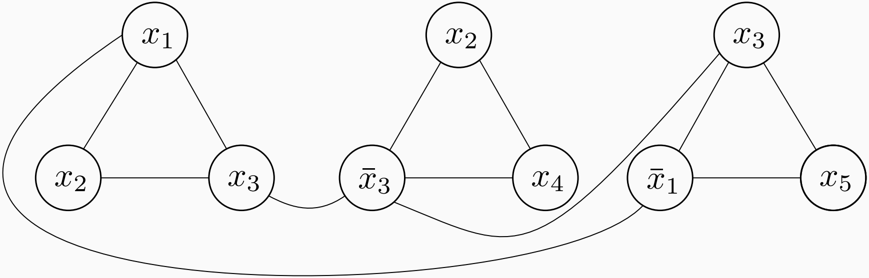
So the 3-CNF has a satisfying assignment if and only if G has an independent set of size k $\square$

Example of the reduction

Consider $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$

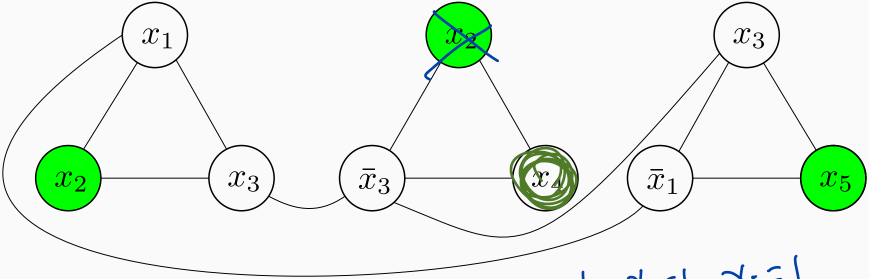
Example of the reduction

Consider $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$



Example of the reduction

Consider $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$



$x_2 = 1$, $x_5 = 1$

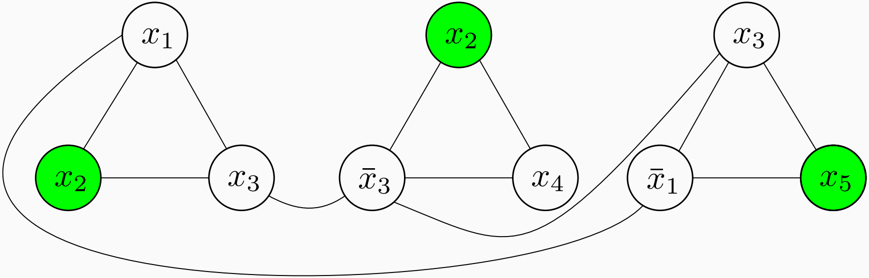
$x_1 = 0$, $x_3 = 0$, $x_4 = 0$

$x_2 = 1$, $x_4 = 1$, $x_5 = 1$

$x_1 = 0$, $x_3 = 0$

Example of the reduction

Consider $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_1 \vee x_5)$



Satisfying assignment: $x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 1$

NP and Computational Hardness

P, NP, and NP-completeness
(Kleinberg-Tardos, Section 8.3, 8.4)

Problems and algorithms

We can encode the input (an instance) of any computational problem as a binary string

Problems and algorithms

We can encode the input (an instance) of any computational problem as a binary string

X_{yes} X_{No}

A **decision problem** X is the set of strings on which the answer is "yes"

Problems and algorithms

We can encode the input (an instance) of any computational problem as a binary string

A **decision problem** X is the set of strings on which the answer is "yes"

An **algorithm** A for a decision problem receives an input string s and outputs $A(s) = \begin{cases} \text{yes} \\ \text{no} \end{cases}$

Problems and algorithms

We can encode the input (an instance) of any computational problem as a binary string

A **decision problem** X is the set of strings on which the answer is "yes"

An **algorithm** A for a decision problem receives an input string s and outputs $A(s) = \begin{cases} \text{yes} \\ \text{no} \end{cases}$

The algorithm A **solves** X if for all s, $A(s) = \text{yes}$ if and only if $s \in \text{X}$

Problems and algorithms

We can encode the input (an instance) of any computational problem as a binary string

A **decision problem** X is the set of strings on which the answer is "yes"

An **algorithm** A for a decision problem receives an input string s and outputs $A(s) = \begin{cases} \text{yes} \\ \text{no} \end{cases}$

The algorithm A **solves** X if for all s, $A(s) = \text{yes}$ if and only if $s \in \text{X}$

The algorithm A has **polynomial running time** if there is a polynomial p s.t. for all s, A terminates on s in at most $O(p(|s|))$ steps

Problems and algorithms

We can encode the input (an instance) of any computational problem as a binary string

A **decision problem** X is the set of strings on which the answer is "yes"

An **algorithm** A for a decision problem receives an input string s and outputs $A(s) = \begin{cases} \text{yes} \\ \text{no} \end{cases}$

The algorithm A **solves** X if for all s, $A(s) = \text{yes}$ if and only if $s \in \text{X}$

The algorithm A has **polynomial running time** if there is a polynomial p s.t. for all s, A terminates on s in at most $O(p(|s|))$ steps

Computational class

P : the class of all problems for which there exists a polynomial-time algorithm

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t and

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \text{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT:

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \le p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT: certificate: an assignment

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \text{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT: certificate: an assignment
  instance s: $(\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT: certificate: an assignment
  instance s: $(\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$
  certificate t: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT: certificate: an assignment
  instance s: $(\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$
  certificate t: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$
- Independent set.

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s,t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT: certificate: an assignment
  instance s: $(\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$
  certificate t: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$
- Independent set. certificate: a set of at least k vertices

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT: certificate: an assignment
  instance s: $(\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$
  certificate t: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$
- Independent set. certificate: a set of at least k vertices
  certifier: check if there's no edge joining them

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT: certificate: an assignment
  instance s: $(\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$
  certificate t: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$
- Independent set. certificate: a set of at least k vertices
  certifier: check if there's no edge joining them

We can use B to design an algorithm for X: use brute force to find a t.

Checking vs solving

Definition

An algorithm B is an **efficient certifier** for a problem X if

- B is a polynomial-time algorithm that takes two inputs s, t, and
- there exists a polynomial p s.t. for all s, we have $s \in \mathrm{X}$ if and only if there exists a string t s.t. $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$

The string t is called a **certificate**

Example:

- 3-SAT: certificate: an assignment
  instance s: $(\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$
  certificate t: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$
- Independent set. certificate: a set of at least k vertices
  certifier: check if there's no edge joining them

We can use B to design an algorithm for X: use brute force to find a t.

But there might be exponentially many possible t's

The computational class NP

Computational class

NP : the class of all problems for which there exists an efficient certifier

The computational class NP

Computational class

NP : the class of all problems for which there exists an efficient certifier

It is easy to see: 3-SAT $\in$ **NP** Independent Set $\in NP$

Vertex Cover $\in NP$

The computational class NP

Computational class

NP : the class of all problems for which there exists an efficient certifier

It is easy to see: 3-SAT $\in$ **NP**

Lemma

P $\subseteq$ **NP**

For any problem in P, there exists an efficient certifier for it.

How to show this?

The computational class NP

Computational class

NP : the class of all problems for which there exists an efficient certifier

It is easy to see: 3-SAT $\in$ **NP**

Lemma

P $\subseteq$ **NP**

Proof.

For any problem in **P** with algorithm A, we construct a certifier B that just returns $A(s)$ with empty certificate t $\square$

NP-completeness

Fundamental question in CS: is $\mathbf{P} = \mathbf{NP}$?

NP-completeness

Fundamental question in CS: is $\mathbf{P} = \mathbf{NP}$? i.e., does there exist a problem $X \in \mathbf{NP}$ but $X \notin \mathbf{P}$?

NP-completeness

Fundamental question in CS: is $\mathbf{P} = \mathbf{NP}$? i.e., does there exist a problem $X \in \mathbf{NP}$ but $X \notin \mathbf{P}$?

We don't know the answer, but we try to find the most difficult problems in **NP**:

Fundamental question in CS: is $\mathbf{P} = \mathbf{NP}$? i.e., does there exist a problem $X \in \mathbf{NP}$ but $X \notin \mathbf{P}$?

We don't know the answer, but we try to find the most difficult problems in **NP**:

Definition

A problem X is **NP-complete** if

NP-completeness

Fundamental question in CS: is $\mathbf{P} = \mathbf{NP}$? i.e., does there exist a problem $X \in \mathbf{NP}$ but $X \notin \mathbf{P}$?

We don't know the answer, but we try to find the most difficult problems in **NP**:

Definition

A problem X is **NP-complete** if

- $X \in \mathbf{NP}$ and

NP-completeness

Fundamental question in CS: is $\mathbf{P} = \mathbf{NP}$? i.e., does there exist a problem $X \in \mathbf{NP}$ but $X \notin \mathbf{P}$?

We don't know the answer, but we try to find the most difficult problems in **NP**:

Definition

A problem X is **NP-complete** if

- $X \in \mathbf{NP}$ and
- for all $Y \in \mathbf{NP}$, $Y \leq_P X$

NP-completeness

Fundamental question in CS: is $\mathbf{P} = \mathbf{NP}$? i.e., does there exist a problem $X \in \mathbf{NP}$ but $X \notin \mathbf{P}$?

We don't know the answer, but we try to find the most difficult problems in **NP**:

Definition

A problem X is **NP-complete** if

- $X \in \mathbf{NP}$ and
- for all $Y \in \mathbf{NP}$, $Y \leq_P X$

Suppose you know

X is NP-complete

and $X \leq_p Y$, then Y is NP-complete

Lemma

If an **NP***-complete problem can be solved in polynomial time, then* $\mathbf{P} = \mathbf{NP}$

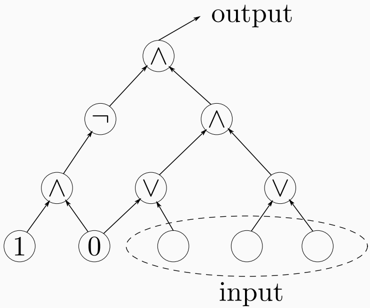
Which problems are NP-complete?

A first **NP**-complete problem: Circuit Satisfiability

Which problems are NP-complete?

A first **NP**-complete problem: Circuit Satisfiability

A circuit consists of

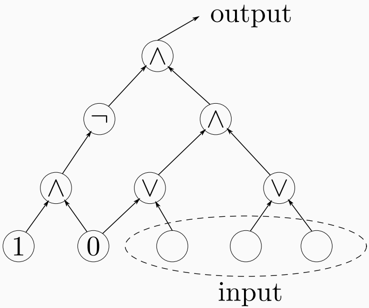


Which problems are NP-complete?

A first **NP**-complete problem: Circuit Satisfiability

A circuit consists of

- inputs

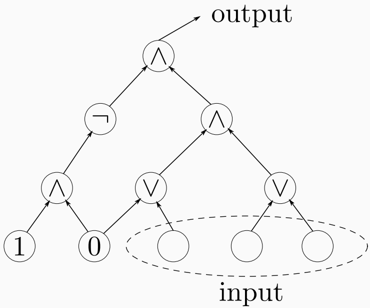


Which problems are NP-complete?

A first **NP**-complete problem: Circuit Satisfiability

A circuit consists of

- inputs
- wires

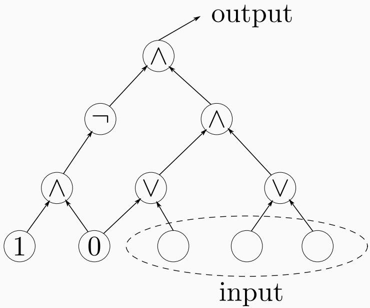


Which problems are NP-complete?

A first **NP**-complete problem: Circuit Satisfiability

A circuit consists of

- inputs
- wires
- logical gates $\vee, \wedge, \neg$ 非门

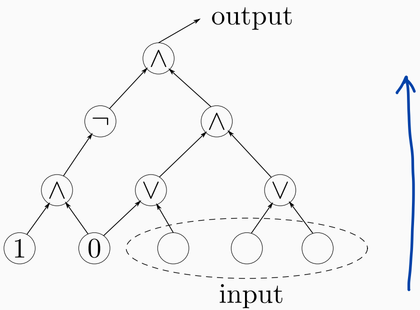


Which problems are NP-complete?

A first **NP**-complete problem: Circuit Satisfiability

A circuit consists of

- inputs
- wires
- logical gates $\vee, \wedge, \neg$
- single output

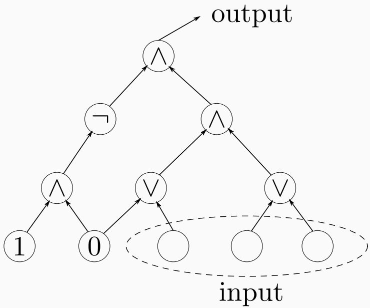


Which problems are NP-complete?

A first **NP**-complete problem: Circuit Satisfiability

A circuit consists of

- inputs
- wires
- logical gates $\vee, \wedge, \neg$
- single output



The Circuit Satisfiability Problem (circuit-SAT)

Instance: A circuit C

Objective: Decide if C is satisfiable

The Cook-Levin Theorem

Theorem (Cook-Levin)

circuit-SAT is **NP***-complete*