The class **NP**

Xiaofeng Gu[1]

[1]Department of Mathematics
West Virginia University

NP-completeness

Outline

Outline

Outline

Class NP

Recall

NP is the class of languages decided by nondeterministic Turing machines in polynomial time.

Definition

Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on strings. R is called **polynomially decidable** if the language $\{x; y : (x, y) \in R\}$ is decided by a deterministic Turing machine in polynomial time.

Definition

R is **polynomial balanced** if $(x, y) \in R$ implies $|y| \leq |x|^k$ for some $k \geq 1$.
That is, the length of the second component is always bounded by a polynomial in the length of the first.

Class NP

Recall

NP is the class of languages decided by nondeterministic Turing machines in polynomial time.

Definition

Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on strings. R is called **polynomially decidable** if the language $\{x; y : (x, y) \in R\}$ is decided by a deterministic Turing machine in polynomial time.

Definition

R is **polynomial balanced** if $(x, y) \in R$ implies $|y| \leq |x|^k$ for some $k \geq 1$.
That is, the length of the second component is always bounded by a polynomial in the length of the first.

Class NP

Recall

NP is the class of languages decided by nondeterministic Turing machines in polynomial time.

Definition

Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on strings. R is called **polynomially decidable** if the language $\{x; y : (x, y) \in R\}$ is decided by a deterministic Turing machine in polynomial time.

Definition

R is **polynomial balanced** if $(x, y) \in R$ implies $|y| \leq |x|^k$ for some $k \geq 1$.
That is, the length of the second component is always bounded by a polynomial in the length of the first.

Class NP (contd.)

Proposition

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomial decidable and polynomial balanced relation R, such that $L = \{x : \exists y, (x, y) \in R\}$.*

Proof.

(i) **"If"** part: Suppose that such an R exists, we need to show L is decided by a nondeterministic Turing machine M in polynomial time. We construct M as follows: On input x, M guesses a y of length at most $|x|^k$, and then verify whether $(x, y) \in R$ (This can be done in polynomial time because R is polynomial decidable.). If $(x, y) \in R$, M accepts, otherwise it rejects.

(ii) **"Only if"** part: Suppose that $L \in$ **NP**, that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$ for some k. Define a relation R as follows: $(x, y) \in R$ if and only if y encodes an accepting computation of N on input x. Clearly R is polynomial decidable and polynomial bounded.
Now we show $L = \{x : \exists y, (x, y) \in R\}$. Since N decides L, $\forall x \in L$, there must be a y such that $(x, y) \in R$, and hence $L \subseteq \{x : \exists y, (x, y) \in R\}$; Conversely, $\forall x \in \{x : \exists y, (x, y) \in R\}$, it must be the cast that N accepts x. It means $x \in L$, and hence $\{x : \exists y, (x, y) \in R\} \subseteq L$. Thus $L = \{x : \exists y, (x, y) \in R\}$.

□

Class NP (contd.)

Proposition

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomial decidable and polynomial balanced relation R, such that $L = \{x : \exists y, (x, y) \in R\}$.*

Proof.

(i) **"If"** part: Suppose that such an R exists, we need to show L is decided by a nondeterministic Turing machine M in polynomial time. We construct M as follows: On input x, M guesses a y of length at most $|x|^k$, and then verify whether $(x, y) \in R$ (This can be done in polynomial time because R is polynomial decidable.). If $(x, y) \in R$, M accepts, otherwise it rejects.

(ii) **"Only if"** part: Suppose that $L \in$ **NP**, that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$ for some k. Define a relation R as follows: $(x, y) \in R$ if and only if y encodes an accepting computation of N on input x. Clearly R is polynomial decidable and polynomial bounded.
Now we show $L = \{x : \exists y, (x, y) \in R\}$. Since N decides L, $\forall x \in L$, there must be a y such that $(x, y) \in R$, and hence $L \subseteq \{x : \exists y, (x, y) \in R\}$; Conversely, $\forall x \in \{x : \exists y, (x, y) \in R\}$, it must be the cast that N accepts x. It means $x \in L$, and hence $\{x : \exists y, (x, y) \in R\} \subseteq L$. Thus $L = \{x : \exists y, (x, y) \in R\}$.

□

Class NP (contd.)

Proposition

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomial decidable and polynomial balanced relation R, such that $L = \{x : \exists y, (x, y) \in R\}$.*

Proof.

(i) **"If"** part: Suppose that such an R exists, we need to show L is decided by a nondeterministic Turing machine M in polynomial time. We construct M as follows: On input x, M guesses a y of length at most $|x|^k$, and then verify whether $(x, y) \in R$ (This can be done in polynomial time because R is polynomial decidable.). If $(x, y) \in R$, M accepts, otherwise it rejects.

(ii) **"Only if"** part: Suppose that $L \in$ **NP**, that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$ for some k. Define a relation R as follows: $(x, y) \in R$ if and only if y encodes an accepting computation of N on input x. Clearly R is polynomial decidable and polynomial bounded.
Now we show $L = \{x : \exists y, (x, y) \in R\}$. Since N decides L, $\forall x \in L$, there must be a y such that $(x, y) \in R$, and hence $L \subseteq \{x : \exists y, (x, y) \in R\}$; Conversely, $\forall x \in \{x : \exists y, (x, y) \in R\}$, it must be the cast that N accepts x. It means $x \in L$, and hence $\{x : \exists y, (x, y) \in R\} \subseteq L$. Thus $L = \{x : \exists y, (x, y) \in R\}$.

□

Class NP (contd.)

Proposition

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomial decidable and polynomial balanced relation R, such that $L = \{x : \exists y, (x, y) \in R\}$.*

Proof.

(i) **"If"** part: Suppose that such an R exists, we need to show L is decided by a nondeterministic Turing machine M in polynomial time. We construct M as follows: On input x, M guesses a y of length at most $|x|^k$, and then verify whether $(x, y) \in R$ (This can be done in polynomial time because R is polynomial decidable.). If $(x, y) \in R$, M accepts, otherwise it rejects.

(ii) **"Only if"** part: Suppose that $L \in$ **NP**, that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$ for some k. Define a relation R as follows: $(x, y) \in R$ if and only if y encodes an accepting computation of N on input x. Clearly R is polynomial decidable and polynomial bounded.
Now we show $L = \{x : \exists y, (x, y) \in R\}$. Since N decides L, $\forall x \in L$, there must be a y such that $(x, y) \in R$, and hence $L \subseteq \{x : \exists y, (x, y) \in R\}$; Conversely, $\forall x \in \{x : \exists y, (x, y) \in R\}$, it must be the cast that N accepts x. It means $x \in L$, and hence $\{x : \exists y, (x, y) \in R\} \subseteq L$. Thus $L = \{x : \exists y, (x, y) \in R\}$.

□

Class NP (contd.)

Proposition

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomial decidable and polynomial balanced relation R, such that $L = \{x : \exists y, (x, y) \in R\}$.*

Proof.

(i) **"If"** part: Suppose that such an R exists, we need to show L is decided by a nondeterministic Turing machine M in polynomial time. We construct M as follows: On input x, M guesses a y of length at most $|x|^k$, and then verify whether $(x, y) \in R$ (This can be done in polynomial time because R is polynomial decidable.). If $(x, y) \in R$, M accepts, otherwise it rejects.

(ii) **"Only if"** part: Suppose that $L \in$ **NP**, that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$ for some k. Define a relation R as follows: $(x, y) \in R$ if and only if y encodes an accepting computation of N on input x. Clearly R is polynomial decidable and polynomial bounded.
Now we show $L = \{x : \exists y, (x, y) \in R\}$. Since N decides L, $\forall x \in L$, there must be a y such that $(x, y) \in R$, and hence $L \subseteq \{x : \exists y, (x, y) \in R\}$; Conversely, $\forall x \in \{x : \exists y, (x, y) \in R\}$, it must be the cast that N accepts x. It means $x \in L$, and hence $\{x : \exists y, (x, y) \in R\} \subseteq L$. Thus $L = \{x : \exists y, (x, y) \in R\}$.

□

Class NP (contd.)

Proposition

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomial decidable and polynomial balanced relation R, such that $L = \{x : \exists y, (x, y) \in R\}$.*

Proof.

(i) **"If"** part: Suppose that such an R exists, we need to show L is decided by a nondeterministic Turing machine M in polynomial time. We construct M as follows: On input x, M guesses a y of length at most $|x|^k$, and then verify whether $(x, y) \in R$ (This can be done in polynomial time because R is polynomial decidable.). If $(x, y) \in R$, M accepts, otherwise it rejects.

(ii) **"Only if"** part: Suppose that $L \in$ **NP**, that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$ for some k. Define a relation R as follows: $(x, y) \in R$ if and only if y encodes an accepting computation of N on input x. Clearly R is polynomial decidable and polynomial bounded.
Now we show $L = \{x : \exists y, (x, y) \in R\}$. Since N decides L, $\forall x \in L$, there must be a y such that $(x, y) \in R$, and hence $L \subseteq \{x : \exists y, (x, y) \in R\}$; Conversely, $\forall x \in \{x : \exists y, (x, y) \in R\}$, it must be the cast that N accepts x. It means $x \in L$, and hence $\{x : \exists y, (x, y) \in R\} \subseteq L$. Thus $L = \{x : \exists y, (x, y) \in R\}$.

□

Class NP (contd.)

Proposition

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomial decidable and polynomial balanced relation R, such that $L = \{x : \exists y, (x, y) \in R\}$.*

Proof.

(i) **"If"** part: Suppose that such an R exists, we need to show L is decided by a nondeterministic Turing machine M in polynomial time. We construct M as follows: On input x, M guesses a y of length at most $|x|^k$, and then verify whether $(x, y) \in R$ (This can be done in polynomial time because R is polynomial decidable.). If $(x, y) \in R$, M accepts, otherwise it rejects.

(ii) **"Only if"** part: Suppose that $L \in$ **NP**, that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$ for some k. Define a relation R as follows: $(x, y) \in R$ if and only if y encodes an accepting computation of N on input x. Clearly R is polynomial decidable and polynomial bounded.
Now we show $L = \{x : \exists y, (x, y) \in R\}$. Since N decides L, $\forall x \in L$, there must be a y such that $(x, y) \in R$, and hence $L \subseteq \{x : \exists y, (x, y) \in R\}$; Conversely, $\forall x \in \{x : \exists y, (x, y) \in R\}$, it must be the cast that N accepts x. It means $x \in L$, and hence $\{x : \exists y, (x, y) \in R\} \subseteq L$. Thus $L = \{x : \exists y, (x, y) \in R\}$.

□

Class NP (contd.)

Proposition

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomial decidable and polynomial balanced relation R, such that $L = \{x : \exists y, (x, y) \in R\}$.*

Proof.

(i) **"If"** part: Suppose that such an R exists, we need to show L is decided by a nondeterministic Turing machine M in polynomial time. We construct M as follows: On input x, M guesses a y of length at most $|x|^k$, and then verify whether $(x, y) \in R$ (This can be done in polynomial time because R is polynomial decidable.). If $(x, y) \in R$, M accepts, otherwise it rejects.

(ii) **"Only if"** part: Suppose that $L \in$ **NP**, that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$ for some k. Define a relation R as follows: $(x, y) \in R$ if and only if y encodes an accepting computation of N on input x. Clearly R is polynomial decidable and polynomial bounded.
Now we show $L = \{x : \exists y, (x, y) \in R\}$. Since N decides L, $\forall x \in L$, there must be a y such that $(x, y) \in R$, and hence $L \subseteq \{x : \exists y, (x, y) \in R\}$; Conversely, $\forall x \in \{x : \exists y, (x, y) \in R\}$, it must be the cast that N accepts x. It means $x \in L$, and hence $\{x : \exists y, (x, y) \in R\} \subseteq L$. Thus $L = \{x : \exists y, (x, y) \in R\}$.

□

What does the proposition tell us?

Note

(i) *Any "yes" instance x of the problem in* **NP** *has at least one polynomial certificate y of its being a "yes" instance.*

(ii) *We may not know how to discover this certificate in polynomial time, but we are sure it exists if the instance is a "yes" instance.*

(iii) *Naturally, "no" instance may not have such certificate.*

Examples

SAT: The certificate is just an assignment that satisfies the Boolean expression.
HAMILTON PATH: the certificate is precisely a Hamilton path in the graph.

What does the proposition tell us?

Note

(i) *Any "yes" instance x of the problem in* **NP** *has at least one polynomial certificate y of its being a "yes" instance.*

(ii) *We may not know how to discover this certificate in polynomial time, but we are sure it exists if the instance is a "yes" instance.*

(iii) *Naturally, "no" instance may not have such certificate.*

Examples

SAT: The certificate is just an assignment that satisfies the Boolean expression.
HAMILTON PATH: the certificate is precisely a Hamilton path in the graph.

What does the proposition tell us?

Note

(i) *Any "yes" instance x of the problem in* **NP** *has at least one polynomial certificate y of its being a "yes" instance.*

(ii) *We may not know how to discover this certificate in polynomial time, but we are sure it exists if the instance is a "yes" instance.*

(iii) *Naturally, "no" instance may not have such certificate.*

Examples

SAT: The certificate is just an assignment that satisfies the Boolean expression.
HAMILTON PATH: the certificate is precisely a Hamilton path in the graph.

What does the proposition tell us?

Note

(i) *Any "yes" instance x of the problem in* **NP** *has at least one polynomial certificate y of its being a "yes" instance.*

(ii) *We may not know how to discover this certificate in polynomial time, but we are sure it exists if the instance is a "yes" instance.*

(iii) *Naturally, "no" instance may not have such certificate.*

Examples

SAT: The certificate is just an assignment that satisfies the Boolean expression.
HAMILTON PATH: the certificate is precisely a Hamilton path in the graph.

What does the proposition tell us?

Note

(i) *Any "yes" instance x of the problem in* **NP** *has at least one polynomial certificate y of its being a "yes" instance.*

(ii) *We may not know how to discover this certificate in polynomial time, but we are sure it exists if the instance is a "yes" instance.*

(iii) *Naturally, "no" instance may not have such certificate.*

Examples

SAT: The certificate is just an assignment that satisfies the Boolean expression.
HAMILTON PATH: the certificate is precisely a Hamilton path in the graph.

Outline

SAT

Recall

(**Cook's Theorem**) SAT is **NP**-complete.

Definition

kSAT, where $k \geq 1$ is an integer, is the special case of SAT in which the formula is in CNF, and all clauses have k literals.

SAT

Recall

(**Cook's Theorem**) SAT is **NP**-complete.

Definition

kSAT, where $k \geq 1$ is an integer, is the special case of SAT in which the formula is in CNF, and all clauses have k literals.

3SAT

Proposition

3SAT *is* **NP**-*complete.*

Proof.

First, it is easy to see that 3SAT $\in$ **NP**. We can construct a nondeterministic Turing machine to guess a truth assignment for the variables and check in polynomial time whether the assignment satisfies all the three-literal clauses.
Then, we can reduce SAT to 3SAT. Suppose c is a k-literal clause in the input CNF expression. If $k = 1$, $c = (x)$,then $c = (x, x, x)$;
If $k = 2$, $c = (x, y)$, then $c = (x, y, y)$;
If $k = 3$, $c = (x, y, z)$;
If $k = 4$, $c = (x_1, x_2, x_3, x_4)$, rewrite as $(x_1, x_2, u) \wedge (x_3, x_4, \bar{u})$.
When $k \geq 4$, $c = (x_1, x_2, x_3, x_4, \ldots, x_k)$, rewrite as
$(x_1, x_2, u_1) \wedge (x_3, \bar{u_1}, u_2) \wedge (x_4, \bar{u_2}, u_3) \wedge \cdots \wedge (x_{k-1}, x_k, \bar{u_{k-3}})$.

□

3SAT

Proposition

3SAT *is* **NP***-complete.*

Proof.

First, it is easy to see that 3SAT $\in$ **NP**. We can construct a nondeterministic Turing machine to guess a truth assignment for the variables and check in polynomial time whether the assignment satisfies all the three-literal clauses.

Then, we can reduce SAT to 3SAT. Suppose c is a k-literal clause in the input CNF expression. If $k = 1$, $c = (x)$,then $c = (x, x, x)$;
If $k = 2$, $c = (x, y)$, then $c = (x, y, y)$;
If $k = 3$, $c = (x, y, z)$;
If $k = 4$, $c = (x_1, x_2, x_3, x_4)$, rewrite as $(x_1, x_2, u) \wedge (x_3, x_4, \bar{u})$.
When $k \geq 4$, $c = (x_1, x_2, x_3, x_4, \ldots, x_k)$, rewrite as
$(x_1, x_2, u_1) \wedge (x_3, \bar{u_1}, u_2) \wedge (x_4, \bar{u_2}, u_3) \wedge \cdots \wedge (x_{k-1}, x_k, \bar{u_{k-3}})$.

□

3SAT

Proposition

3SAT *is* **NP**-*complete.*

Proof.

First, it is easy to see that 3SAT $\in$ **NP**. We can construct a nondeterministic Turing machine to guess a truth assignment for the variables and check in polynomial time whether the assignment satisfies all the three-literal clauses.
Then, we can reduce SAT to 3SAT. Suppose c is a k-literal clause in the input CNF expression. If $k = 1$, $c = (x)$,then $c = (x, x, x)$;
If $k = 2$, $c = (x, y)$, then $c = (x, y, y)$;
If $k = 3$, $c = (x, y, z)$;
If $k = 4$, $c = (x_1, x_2, x_3, x_4)$, rewrite as $(x_1, x_2, u) \wedge (x_3, x_4, \bar{u})$.
When $k \geq 4$, $c = (x_1, x_2, x_3, x_4, \ldots, x_k)$, rewrite as
$(x_1, x_2, u_1) \wedge (x_3, \bar{u_1}, u_2) \wedge (x_4, \bar{u_2}, u_3) \wedge \cdots \wedge (x_{k-1}, x_k, \bar{u_{k-3}})$.

□

3SAT

Proposition

3SAT *is* **NP**-*complete.*

Proof.

First, it is easy to see that 3SAT $\in$ **NP**. We can construct a nondeterministic Turing machine to guess a truth assignment for the variables and check in polynomial time whether the assignment satisfies all the three-literal clauses.
Then, we can reduce SAT to 3SAT. Suppose c is a k-literal clause in the input CNF expression. If $k = 1$, $c = (x)$,then $c = (x, x, x)$;
If $k = 2$, $c = (x, y)$, then $c = (x, y, y)$;
If $k = 3$, $c = (x, y, z)$;
If $k = 4$, $c = (x_1, x_2, x_3, x_4)$, rewrite as $(x_1, x_2, u) \wedge (x_3, x_4, \bar{u})$.
When $k \geq 4$, $c = (x_1, x_2, x_3, x_4, \ldots, x_k)$, rewrite as
$(x_1, x_2, u_1) \wedge (x_3, \bar{u_1}, u_2) \wedge (x_4, \bar{u_2}, u_3) \wedge \cdots \wedge (x_{k-1}, x_k, \bar{u_{k-3}})$.

□

3SAT

Proposition

3SAT *is* **NP**-*complete.*

Proof.

First, it is easy to see that 3SAT $\in$ **NP**. We can construct a nondeterministic Turing machine to guess a truth assignment for the variables and check in polynomial time whether the assignment satisfies all the three-literal clauses.
Then, we can reduce SAT to 3SAT. Suppose c is a k-literal clause in the input CNF expression. If $k = 1$, $c = (x)$,then $c = (x, x, x)$;
If $k = 2$, $c = (x, y)$, then $c = (x, y, y)$;
If $k = 3$, $c = (x, y, z)$;
If $k = 4$, $c = (x_1, x_2, x_3, x_4)$, rewrite as $(x_1, x_2, u) \wedge (x_3, x_4, \bar{u})$.
When $k \geq 4$, $c = (x_1, x_2, x_3, x_4, \ldots, x_k)$, rewrite as
$(x_1, x_2, u_1) \wedge (x_3, \bar{u_1}, u_2) \wedge (x_4, \bar{u_2}, u_3) \wedge \cdots \wedge (x_{k-1}, x_k, \bar{u_{k-3}})$.

3SAT

Proposition

3SAT *is* **NP**-*complete.*

Proof.

First, it is easy to see that 3SAT $\in$ **NP**. We can construct a nondeterministic Turing machine to guess a truth assignment for the variables and check in polynomial time whether the assignment satisfies all the three-literal clauses.
Then, we can reduce SAT to 3SAT. Suppose c is a k-literal clause in the input CNF expression. If $k = 1$, $c = (x)$,then $c = (x, x, x)$;
If $k = 2$, $c = (x, y)$, then $c = (x, y, y)$;
If $k = 3$, $c = (x, y, z)$;
If $k = 4$, $c = (x_1, x_2, x_3, x_4)$, rewrite as $(x_1, x_2, u) \wedge (x_3, x_4, \bar{u})$.
When $k \geq 4$, $c = (x_1, x_2, x_3, x_4, \ldots, x_k)$, rewrite as
$(x_1, x_2, u_1) \wedge (x_3, \bar{u_1}, u_2) \wedge (x_4, \bar{u_2}, u_3) \wedge \cdots \wedge (x_{k-1}, x_k, \bar{u_{k-3}})$.

□

3SAT

Proposition

3SAT *is* **NP***-complete.*

Proof.

First, it is easy to see that 3SAT $\in$ **NP**. We can construct a nondeterministic Turing machine to guess a truth assignment for the variables and check in polynomial time whether the assignment satisfies all the three-literal clauses.
Then, we can reduce SAT to 3SAT. Suppose c is a k-literal clause in the input CNF expression. If $k = 1$, $c = (x)$,then $c = (x, x, x)$;
If $k = 2$, $c = (x, y)$, then $c = (x, y, y)$;
If $k = 3$, $c = (x, y, z)$;
If $k = 4$, $c = (x_1, x_2, x_3, x_4)$, rewrite as $(x_1, x_2, u) \wedge (x_3, x_4, \bar{u})$.
When $k \geq 4$, $c = (x_1, x_2, x_3, x_4, \ldots, x_k)$, rewrite as
$(x_1, x_2, u_1) \wedge (x_3, \bar{u_1}, u_2) \wedge (x_4, \bar{u_2}, u_3) \wedge \cdots \wedge (x_{k-1}, x_k, \bar{u_{k-3}})$.

kSAT

Note

In analyzing the complexity of a problem, we are trying to define the precise boundary between the polynomial and **NP***-complete cases.*
For SAT, we already know that 3SAT is **NP***-complete, how about 2SAT?*

kSAT

Note

In analyzing the complexity of a problem, we are trying to define the precise boundary between the polynomial and **NP***-complete cases.*
For SAT*, we already know that* 3SAT *is* **NP***-complete, how about* 2SAT*?*

Outline

2SAT and Graph $G(\phi)$

Definition

Let ϕ be an instance of 2SAT. We define a graph $G(\phi)$ as follows:
(a) The vertices of G are the variables of ϕ and their negations;
(b) There is an edge (α, β) if and only if there is a clause $(\neg\alpha \vee \beta)$ (or $(\beta \vee \neg\alpha)$in ϕ.
(c) $G(\phi)$ has a weird symmetry: If (α, β) is an edge, then so is $(\neg\beta, \neg\alpha)$.

Examples

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$
$G(\phi)$?

2SAT and Graph $G(\phi)$

Definition

Let ϕ be an instance of 2SAT. We define a graph $G(\phi)$ as follows:
(a) The vertices of G are the variables of ϕ and their negations;
(b) There is an edge (α, β) if and only if there is a clause $(\neg\alpha \vee \beta)$ (or $(\beta \vee \neg\alpha)$in ϕ.
(c) $G(\phi)$ has a weird symmetry: If (α, β) is an edge, then so is $(\neg\beta, \neg\alpha)$.

Examples

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$
$G(\phi)$?

2SAT and Graph $G(\phi)$

Definition

Let ϕ be an instance of 2SAT. We define a graph $G(\phi)$ as follows:
(a) The vertices of G are the variables of ϕ and their negations;
(b) There is an edge (α, β) if and only if there is a clause $(\neg\alpha \vee \beta)$ (or $(\beta \vee \neg\alpha)$in ϕ.
(c) $G(\phi)$ has a weird symmetry: If (α, β) is an edge, then so is $(\neg\beta, \neg\alpha)$.

Examples

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$
$G(\phi)$?

2SAT and Graph $G(\phi)$

Definition

Let ϕ be an instance of 2SAT. We define a graph $G(\phi)$ as follows:
(a) The vertices of G are the variables of ϕ and their negations;
(b) There is an edge (α, β) if and only if there is a clause $(\neg\alpha \vee \beta)$ (or $(\beta \vee \neg\alpha)$in ϕ.
(c) $G(\phi)$ has a weird symmetry: If (α, β) is an edge, then so is $(\neg\beta, \neg\alpha)$.

Examples

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$
$G(\phi)$?

2SAT and Graph $G(\phi)$

Definition

Let ϕ be an instance of 2SAT. We define a graph $G(\phi)$ as follows:
(a) The vertices of G are the variables of ϕ and their negations;
(b) There is an edge (α, β) if and only if there is a clause $(\neg\alpha \vee \beta)$ (or $(\beta \vee \neg\alpha)$in ϕ.
(c) $G(\phi)$ has a weird symmetry: If (α, β) is an edge, then so is $(\neg\beta, \neg\alpha)$.

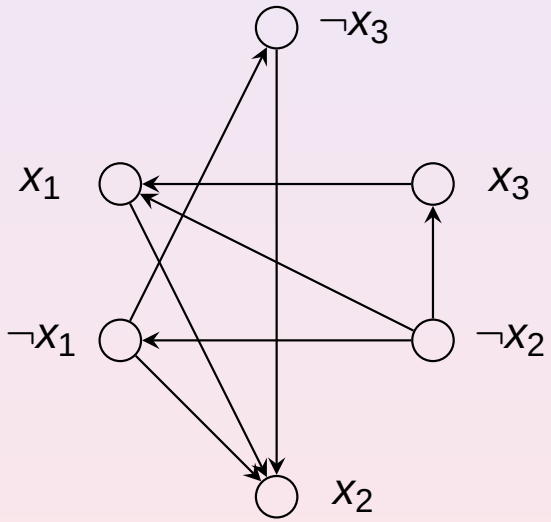
Examples

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$
$G(\phi)$?

2SAT and Graph $G(\phi)$ (contd.)

Example

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



2SAT and Graph $G(\phi)$ (contd.)

Theorem

ϕ is unsatisfiable if and only if there is a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in $G(\phi)$.

Proof.

"If" part: Suppose that such a x exists, we want to show ϕ is unsatisfiable. If ϕ is satisfied by an assignment T, we have two cases:

(a) $T(x) =$ **true**. There is a path from x to $\neg x$, and $T(x) =$ **true** and $T(\neg x) =$ **false**, then there must be an edge (α, β) along this path such that $T(\alpha) =$ **true** and $T(\beta) =$ **false**. Since (α, β) is an edge in $G(\phi)$, $\neg\alpha \vee \beta$ is a clause in ϕ, which is not satisfied by T, a contradiction.

(b) $T(x) =$ **false**. Use path $\neg x$ to x, and the same argument as (a).

□

2SAT and Graph $G(\phi)$ (contd.)

Theorem

ϕ is unsatisfiable if and only if there is a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in $G(\phi)$.

Proof.

"If" part: Suppose that such a x exists, we want to show ϕ is unsatisfiable. If ϕ is satisfied by an assignment T, we have two cases:

(a) $T(x) =$ **true**. There is a path from x to $\neg x$, and $T(x) =$ **true** and $T(\neg x) =$ **false**, then there must be an edge (α, β) along this path such that $T(\alpha) =$ **true** and $T(\beta) =$ **false**. Since (α, β) is an edge in $G(\phi)$, $\neg\alpha \vee \beta$ is a clause in ϕ, which is not satisfied by T, a contradiction.

(b) $T(x) =$ **false**. Use path $\neg x$ to x, and the same argument as (a).

□

2SAT and Graph $G(\phi)$ (contd.)

Theorem

ϕ is unsatisfiable if and only if there is a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in $G(\phi)$.

Proof.

"If" part: Suppose that such a x exists, we want to show ϕ is unsatisfiable. If ϕ is satisfied by an assignment T, we have two cases:

(a) $T(x) =$ **true**. There is a path from x to $\neg x$, and $T(x) =$ **true** and $T(\neg x) =$ **false**, then there must be an edge (α, β) along this path such that $T(\alpha) =$ **true** and $T(\beta) =$ **false**. Since (α, β) is an edge in $G(\phi)$, $\neg\alpha \vee \beta$ is a clause in ϕ, which is not satisfied by T, a contradiction.

(b) $T(x) =$ **false**. Use path $\neg x$ to x, and the same argument as (a).

□

2SAT and Graph $G(\phi)$ (contd.)

Theorem

ϕ is unsatisfiable if and only if there is a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in $G(\phi)$.

Proof.

"If" part: Suppose that such a x exists, we want to show ϕ is unsatisfiable. If ϕ is satisfied by an assignment T, we have two cases:

(a) $T(x) =$ **true**. There is a path from x to $\neg x$, and $T(x) =$ **true** and $T(\neg x) =$ **false**, then there must be an edge (α, β) along this path such that $T(\alpha) =$ **true** and $T(\beta) =$ **false**. Since (α, β) is an edge in $G(\phi)$, $\neg\alpha \vee \beta$ is a clause in ϕ, which is not satisfied by T, a contradiction.

(b) $T(x) =$ **false**. Use path $\neg x$ to x, and the same argument as (a).

□

Proof (contd.)

Proof.

"Only if" part: Suppose that ϕ is unsatisfiable, we want to show there is such a variable x. If there is no such an x, we are going to construct a satisfying assignment, and then by contradiction we prove it.

(a) For a node α, if there is a path from α to $\neg\alpha$, then α must be assigned **false**.

(b) If there's no path from α to $\neg\alpha$, then all nodes that reachable from α are assigned **true**, and all nodes from which $\neg\alpha$ is reachable are assigned **false**.

Repeat the step until all nodes have assignments, we can get a satisfying assignment. We have two problems:

(i) Is the step in (b) well-defined? Yes!

(ii) Continue doing the steps, we will get an assignment. Why is it a satisfying assignment?

□

Proof (contd.)

Proof.

"Only if" part: Suppose that ϕ is unsatisfiable, we want to show there is such a variable x. If there is no such an x, we are going to construct a satisfying assignment, and then by contradiction we prove it.

(a) For a node α, if there is a path from α to $\neg\alpha$, then α must be assigned **false**.

(b) If there's no path from α to $\neg\alpha$, then all nodes that reachable from α are assigned **true**, and all nodes from which $\neg\alpha$ is reachable are assigned **false**.

Repeat the step until all nodes have assignments, we can get a satisfying assignment.

We have two problems:

(i) Is the step in (b) well-defined? Yes!

(ii) Continue doing the steps, we will get an assignment. Why is it a satisfying assignment?

□

Proof (contd.)

Proof.

"Only if" part: Suppose that ϕ is unsatisfiable, we want to show there is such a variable x. If there is no such an x, we are going to construct a satisfying assignment, and then by contradiction we prove it.

(a) For a node α, if there is a path from α to $\neg\alpha$, then α must be assigned **false**.

(b) If there's no path from α to $\neg\alpha$, then all nodes that reachable from α are assigned **true**, and all nodes from which $\neg\alpha$ is reachable are assigned **false**.

Repeat the step until all nodes have assignments, we can get a satisfying assignment. We have two problems:

(i) Is the step in (b) well-defined? Yes!

(ii) Continue doing the steps, we will get an assignment. Why is it a satisfying assignment?

Corollary

2SAT *is in* **NL** *(and therefore in* **P)**.

Proof.

NL is closed under complement.
We can recognize unsatisfiable expressions in **NL**: Guess a variable x, and paths from x to $\neg x$ and back in nondeterministic logarithmic space.

□

Corollary

2SAT *is in* **NL** *(and therefore in* **P)**.

Proof.

NL is closed under complement.
We can recognize unsatisfiable expressions in **NL**: Guess a variable x, and paths from x to $\neg x$ and back in nondeterministic logarithmic space.

□

Outline

MAXkSAT

Definition

We are given a set of clauses, each with two literals in it, and an integer K. MAX2SAT is the problem whether there is an assignment that satisfies at least K of the clauses.

Observation

When $k \geq 3$, MAXkSAT is obviously **NP**-complete.

MAXkSAT

Definition

We are given a set of clauses, each with two literals in it, and an integer K. MAX2SAT is the problem whether there is an assignment that satisfies at least K of the clauses.

Observation

When $k \geq 3$, MAXkSAT is obviously **NP**-complete.

MAX2SAT

Theorem

MAX2SAT *is* **NP**-*complete.*

Proof.

Let us consider a small instance first: given ten clauses

$$(x), (y), (z), (w)$$
$$(\neg x \vee \neg y), (\neg y \vee \neg z), (\neg z \vee \neg x)$$
$$(x \vee \neg w), (y \vee \neg w), (z \vee \neg w)$$

How many clauses can be satisfied? If an assignment satisfy $(x \vee y \vee z)$, then it can be extended to satisfy seven of the clauses and no more. Then 3SAT can be reduced to MAX2SAT: given any instance ϕ of 3SAT, we can construct an instance $R(\phi)$ of MAX2SAT: for each clause $C_i = (\alpha \vee \beta \vee \gamma)$ of ϕ, we add to $R(\phi)$ the following ten clauses:

$$(\alpha), (\beta), (\gamma), (w_i)$$
$$(\neg\alpha \vee \neg\beta), (\neg\beta \vee \neg\gamma), (\neg\gamma \vee \neg\alpha)$$

□

MAX2SAT

Theorem

MAX2SAT *is* **NP**-*complete.*

Proof.

Let us consider a small instance first: given ten clauses

$$(x), (y), (z), (w)$$
$$(\neg x \vee \neg y), (\neg y \vee \neg z), (\neg z \vee \neg x)$$
$$(x \vee \neg w), (y \vee \neg w), (z \vee \neg w)$$

How many clauses can be satisfied? If an assignment satisfy $(x \vee y \vee z)$, then it can be extended to satisfy seven of the clauses and no more. Then 3SAT can be reduced to MAX2SAT: given any instance ϕ of 3SAT, we can construct an instance $R(\phi)$ of MAX2SAT: for each clause $C_i = (\alpha \vee \beta \vee \gamma)$ of ϕ, we add to $R(\phi)$ the following ten clauses:

$$(\alpha), (\beta), (\gamma), (w_i)$$
$$(\neg\alpha \vee \neg\beta), (\neg\beta \vee \neg\gamma), (\neg\gamma \vee \neg\alpha)$$

□

MAX2SAT

Theorem

MAX2SAT *is* **NP**-*complete.*

Proof.

Let us consider a small instance first: given ten clauses

$$(x), (y), (z), (w)$$
$$(\neg x \vee \neg y), (\neg y \vee \neg z), (\neg z \vee \neg x)$$
$$(x \vee \neg w), (y \vee \neg w), (z \vee \neg w)$$

How many clauses can be satisfied? If an assignment satisfy $(x \vee y \vee z)$, then it can be extended to satisfy seven of the clauses and no more. Then 3SAT can be reduced to MAX2SAT: given any instance ϕ of 3SAT, we can construct an instance $R(\phi)$ of MAX2SAT: for each clause $C_i = (\alpha \vee \beta \vee \gamma)$ of ϕ, we add to $R(\phi)$ the following ten clauses:

$$(\alpha), (\beta), (\gamma), (w_i)$$
$$(\neg\alpha \vee \neg\beta), (\neg\beta \vee \neg\gamma), (\neg\gamma \vee \neg\alpha)$$

□

MAX2SAT

Theorem

MAX2SAT *is* **NP**-*complete.*

Proof.

Let us consider a small instance first: given ten clauses

$$(x), (y), (z), (w)$$
$$(\neg x \vee \neg y), (\neg y \vee \neg z), (\neg z \vee \neg x)$$
$$(x \vee \neg w), (y \vee \neg w), (z \vee \neg w)$$

How many clauses can be satisfied? If an assignment satisfy $(x \vee y \vee z)$, then it can be extended to satisfy seven of the clauses and no more. Then 3SAT can be reduced to MAX2SAT: given any instance ϕ of 3SAT, we can construct an instance $R(\phi)$ of MAX2SAT: for each clause $C_i = (\alpha \vee \beta \vee \gamma)$ of ϕ, we add to $R(\phi)$ the following ten clauses:

$$(\alpha), (\beta), (\gamma), (w_i)$$
$$(\neg\alpha \vee \neg\beta), (\neg\beta \vee \neg\gamma), (\neg\gamma \vee \neg\alpha)$$

□

MAX2SAT (contd.)

Proof.

$$(\alpha \vee \neg w_i), (\beta \vee \neg w_i), (\gamma \vee \neg w_i)$$

If ϕ has m clauses, then $R(\phi)$ has $10m$. Set $K = 7m$. We claim that: ϕ is satisfiable if and only there are at least K clauses can be satisfied in $R(\phi)$. □

MAX2SAT (contd.)

Proof.

$$(\alpha \vee \neg w_i), (\beta \vee \neg w_i), (\gamma \vee \neg w_i)$$

If ϕ has m clauses, then $R(\phi)$ has $10m$. Set $K = 7m$. We claim that: ϕ is satisfiable if and only there are at least K clauses can be satisfied in $R(\phi)$. □

Outline

NAESAT

Definition

NAESAT: A Boolean expression in CNF is said to be NAE-satisfied, if in each clause at least one literal is **true** and at least one literal is **false**.

Theorem

NAESAT *is* **NP***-complete.*

Proof.

In Theorem 8.2, we have proved CIRCUIT SAT is **NP**-complete. Now we reduce CIRCUIT SAT to NAESAT, as Example 8.3 on how to reduce CIRCUIT SAT to SAT. We add to all one- or two-literal clauses the same literal, call it z. We claim that the resulting set of clauses are NAE-satisfiable if and only if the original circuit is satisfiable. Suppose that there is an assignment T that NAE-satisfies all clauses. Then $\bar{T}$ also NAE-satisfies all clauses. In one of these assignments z takes the value **false**. This assignment then satisfies all original clauses (before the addition of z) and therefore there is a satisfying assignment for the circuit.
Conversely, it there is an assignment that satisfies the circuit. Then there is an assignment T that satisfies all clauses. We add z and set z **false** in T, then in no clause all literals are **true** under T. Hence, the resulting clauses are NAE-satisfied under T.

NAESAT

Definition

NAESAT: A Boolean expression in CNF is said to be NAE-satisfied, if in each clause at least one literal is **true** and at least one literal is **false**.

Theorem

NAESAT *is* **NP**-*complete.*

Proof.

In Theorem 8.2, we have proved CIRCUIT SAT is **NP**-complete. Now we reduce CIRCUIT SAT to NAESAT, as Example 8.3 on how to reduce CIRCUIT SAT to SAT.
We add to all one- or two-literal clauses the same literal, call it z. We claim that the resulting set of clauses are NAE-satisfiable if and only if the original circuit is satisfiable.
Suppose that there is an assignment T that NAE-satisfies all clauses. Then $\bar{T}$ also NAE-satisfies all clauses. In one of these assignments z takes the value **false**. This assignment then satisfies all original clauses (before the addition of z) and therefore there is a satisfying assignment for the circuit.
Conversely, it there is an assignment that satisfies the circuit. Then there is an assignment T that satisfies all clauses. We add z and set z **false** in T, then in no clause all literals are **true** under T. Hence, the resulting clauses are NAE-satisfied under T.

NAESAT

Definition

NAESAT: A Boolean expression in CNF is said to be NAE-satisfied, if in each clause at least one literal is **true** and at least one literal is **false**.

Theorem

NAESAT *is* **NP**-*complete.*

Proof.

In Theorem 8.2, we have proved CIRCUIT SAT is **NP**-complete. Now we reduce CIRCUIT SAT to NAESAT, as Example 8.3 on how to reduce CIRCUIT SAT to SAT.
We add to all one- or two-literal clauses the same literal, call it z. We claim that the resulting set of clauses are NAE-satisfiable if and only if the original circuit is satisfiable.
Suppose that there is an assignment T that NAE-satisfies all clauses. Then $\bar{T}$ also NAE-satisfies all clauses. In one of these assignments z takes the value **false**. This assignment then satisfies all original clauses (before the addition of z) and therefore there is a satisfying assignment for the circuit.
Conversely, it there is an assignment that satisfies the circuit. Then there is an assignment T that satisfies all clauses. We add z and set z **false** in T, then in no clause all literals are **true** under T. Hence, the resulting clauses are NAE-satisfied under T.

NAESAT

Definition

NAESAT: A Boolean expression in CNF is said to be NAE-satisfied, if in each clause at least one literal is **true** and at least one literal is **false**.

Theorem

NAESAT *is* **NP**-*complete.*

Proof.

In Theorem 8.2, we have proved CIRCUIT SAT is **NP**-complete. Now we reduce CIRCUIT SAT to NAESAT, as Example 8.3 on how to reduce CIRCUIT SAT to SAT. We add to all one- or two-literal clauses the same literal, call it z. We claim that the resulting set of clauses are NAE-satisfiable if and only if the original circuit is satisfiable.

Suppose that there is an assignment T that NAE-satisfies all clauses. Then $\bar{T}$ also NAE-satisfies all clauses. In one of these assignments z takes the value **false**. This assignment then satisfies all original clauses (before the addition of z) and therefore there is a satisfying assignment for the circuit.

Conversely, it there is an assignment that satisfies the circuit. Then there is an assignment T that satisfies all clauses. We add z and set z **false** in T, then in no clause all literals are **true** under T. Hence, the resulting clauses are NAE-satisfied under T.

NAESAT

Definition

NAESAT: A Boolean expression in CNF is said to be NAE-satisfied, if in each clause at least one literal is **true** and at least one literal is **false**.

Theorem

NAESAT *is* **NP**-*complete.*

Proof.

In Theorem 8.2, we have proved CIRCUIT SAT is **NP**-complete. Now we reduce CIRCUIT SAT to NAESAT, as Example 8.3 on how to reduce CIRCUIT SAT to SAT. We add to all one- or two-literal clauses the same literal, call it z. We claim that the resulting set of clauses are NAE-satisfiable if and only if the original circuit is satisfiable. Suppose that there is an assignment T that NAE-satisfies all clauses. Then $\bar{T}$ also NAE-satisfies all clauses. In one of these assignments z takes the value **false**. This assignment then satisfies all original clauses (before the addition of z) and therefore there is a satisfying assignment for the circuit.

Conversely, it there is an assignment that satisfies the circuit. Then there is an assignment T that satisfies all clauses. We add z and set z **false** in T, then in no clause all literals are **true** under T. Hence, the resulting clauses are NAE-satisfied under T.

NAESAT

Definition

NAESAT: A Boolean expression in CNF is said to be NAE-satisfied, if in each clause at least one literal is **true** and at least one literal is **false**.

Theorem

NAESAT *is* **NP**-*complete.*

Proof.

In Theorem 8.2, we have proved CIRCUIT SAT is **NP**-complete. Now we reduce CIRCUIT SAT to NAESAT, as Example 8.3 on how to reduce CIRCUIT SAT to SAT. We add to all one- or two-literal clauses the same literal, call it z. We claim that the resulting set of clauses are NAE-satisfiable if and only if the original circuit is satisfiable. Suppose that there is an assignment T that NAE-satisfies all clauses. Then $\bar{T}$ also NAE-satisfies all clauses. In one of these assignments z takes the value **false**. This assignment then satisfies all original clauses (before the addition of z) and therefore there is a satisfying assignment for the circuit.
Conversely, it there is an assignment that satisfies the circuit. Then there is an assignment T that satisfies all clauses. We add z and set z **false** in T, then in no clause all literals are **true** under T. Hence, the resulting clauses are NAE-satisfied under T.

Outline

INDEPENDENT SET

Definition

Let $G = (V, E)$ be an undirected graph, and $V' \subseteq V$. We say that V' is independent if $\forall i, j \in V', (i, j) \notin E$.

Definition

INDEPENDENT SET problem: Given an undirected graph and an integer K, is there an independent set V' with $|V'| \geq K$?

INDEPENDENT SET

Definition

Let $G = (V, E)$ be an undirected graph, and $V' \subseteq V$. We say that V' is independent if $\forall i, j \in V', (i, j) \notin E$.

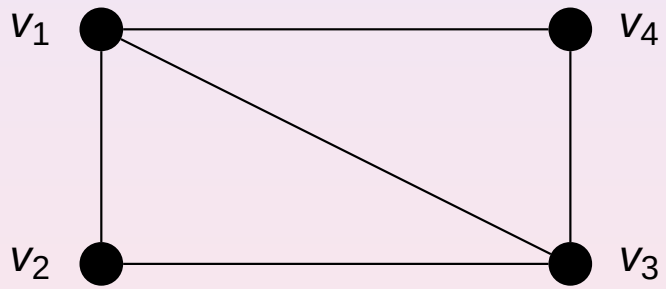
Definition

INDEPENDENT SET problem: Given an undirected graph and an integer K, is there an independent set V' with $|V'| \geq K$?

INDEPENDENT SET (contd.)

Example

In the graph below, $V' = \{v_2, v_4\}$ is an independent set.



INDEPENDENT SET (contd.)

Theorem

INDEPENDENT SET *is* **NP**-*complete.*

Proof.

Reduce 3SAT to INDEPENDENT SET.
Given an instance ϕ of 3SAT with m clauses. We can construct a graph $R(\phi)$:
(a) For each one of the m clauses, we create a separate triangle in the graph;
(b) Each node of the triangle corresponds to a literal in the clause;
(c) There is an edge between two nodes u and v in different triangles if and only if $v = \neg u$.

□

INDEPENDENT SET (contd.)

Theorem

INDEPENDENT SET *is* **NP***-complete.*

Proof.

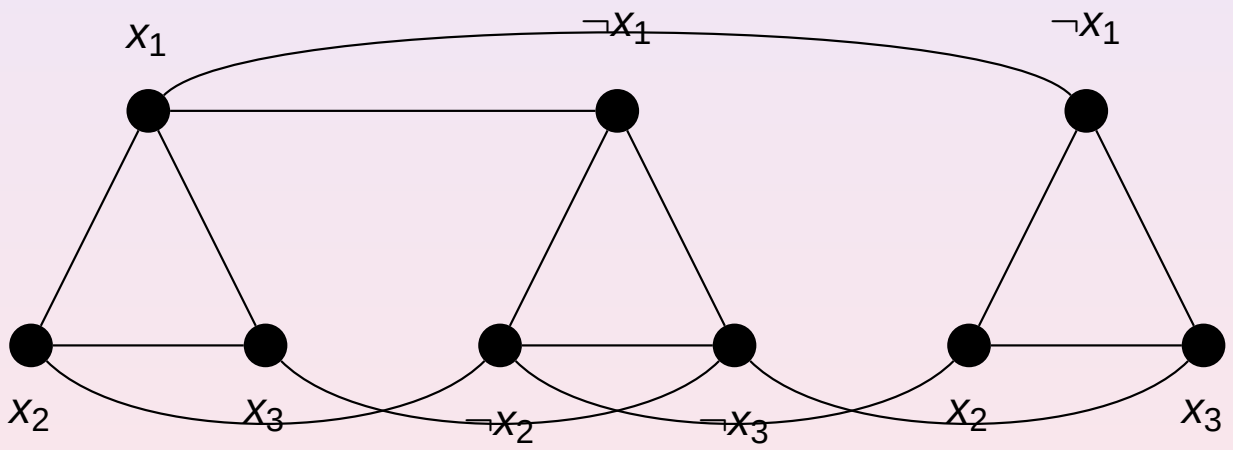
Reduce 3SAT to INDEPENDENT SET.
Given an instance ϕ of 3SAT with m clauses. We can construct a graph $R(\phi)$:
(a) For each one of the m clauses, we create a separate triangle in the graph;
(b) Each node of the triangle corresponds to a literal in the clause;
(c) There is an edge between two nodes u and v in different triangles if and only if $v = \neg u$.

□

Proof (contd.)

Example

$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



Proof (contd.)

Proof.

m clauses correspond m triangles. Set $K = m$. We claim that ϕ is satisfiable if and only if there is an independent set V' of K nodes in graph $R(\phi)$. To see this, if a satisfying assignment exists, then we identify a true literal in each clause, and pick the node in the triangle of this clause labeled by this literal; Conversely, if sucha set V' exists, just set the vertices in V' to be **true** and then we can get a satisfying assignment.

□

Proof (contd.)

Proof.

m clauses correspond m triangles. Set $K = m$. We claim that ϕ is satisfiable if and only if there is an independent set V' of K nodes in graph $R(\phi)$. To see this, if a satisfying assignment exists, then we identify a true literal in each clause, and pick the node in the triangle of this clause labeled by this literal; Conversely, if sucha set V' exists, just set the vertices in V' to be **true** and then we can get a satisfying assignment.

□

Proof (contd.)

Proof.

m clauses correspond m triangles. Set $K = m$. We claim that ϕ is satisfiable if and only if there is an independent set V' of K nodes in graph $R(\phi)$. To see this, if a satisfying assignment exists, then we identify a true literal in each clause, and pick the node in the triangle of this clause labeled by this literal; Conversely, if sucha set V' exists, just set the vertices in V' to be **true** and then we can get a satisfying assignment.

□

Application of INDEPENDENT SET: CLIQUE

Definition

A clique in an undirected graph is a set of pairwise adjacent vertices.

Definition

CLIQUE problem: Given an undirected graph G and an integer K, whether there is a set of K vertices that form a clique by having all possible edges between them?

Corollary

CLIQUE *is* **NP***-complete.*

Proof.

Outline of proof:
vertex subset C is a clique in a graph G if and only if it is an independent set in G^c, the complement of G.

□

Application of INDEPENDENT SET: CLIQUE

Definition

A clique in an undirected graph is a set of pairwise adjacent vertices.

Definition

CLIQUE problem: Given an undirected graph G and an integer K, whether there is a set of K vertices that form a clique by having all possible edges between them?

Corollary

CLIQUE *is* **NP***-complete.*

Proof.

Outline of proof:
vertex subset C is a clique in a graph G if and only if it is an independent set in G^c, the complement of G.

□

Application of INDEPENDENT SET: CLIQUE

Definition

A clique in an undirected graph is a set of pairwise adjacent vertices.

Definition

CLIQUE problem: Given an undirected graph G and an integer K, whether there is a set of K vertices that form a clique by having all possible edges between them?

Corollary

CLIQUE *is* **NP***-complete.*

Proof.

Outline of proof:
vertex subset C is a clique in a graph G if and only if it is an independent set in G^c, the complement of G.

□

Application of INDEPENDENT SET: CLIQUE

Definition

A clique in an undirected graph is a set of pairwise adjacent vertices.

Definition

CLIQUE problem: Given an undirected graph G and an integer K, whether there is a set of K vertices that form a clique by having all possible edges between them?

Corollary

CLIQUE *is* **NP***-complete.*

Proof.

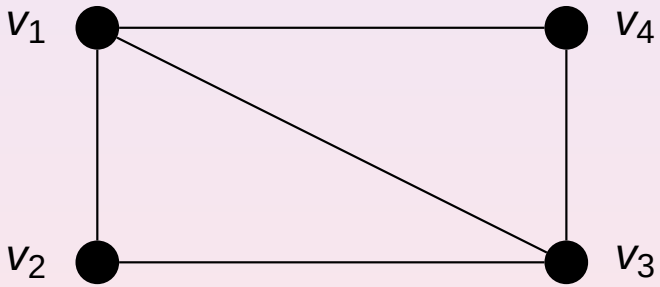
Outline of proof:
vertex subset C is a clique in a graph G if and only if it is an independent set in G^c, the complement of G.

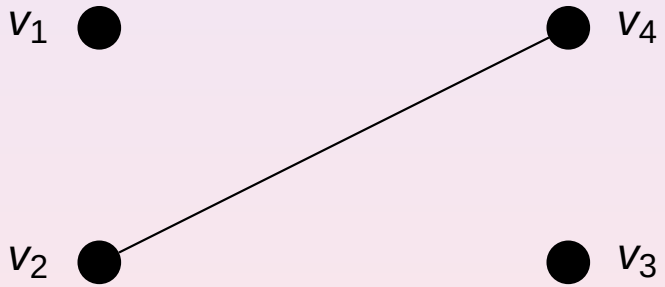
□

CLIQUE (contd.)

Example

$C = \{v_1, v_2, v_3\}$ is a clique in the first graph, and also it is a independent set in the second graph, which is the complement of the first graph.





Application of INDEPENDENT SET: NODE COVER

Definition

A node cover of an undirected graph $G = (V, E)$ is a set $C \subseteq V$ that contains at least one endpoint of every edge.

Definition

NODE COVER problem: Given a graph and an integer K, whether there is a node cover C with K or fewer vertices?

Corollary

NODE COVER *is* **NP***-complete.*

Proof.

Outline of proof:
Vertex subset C is a node cover of a graph G if and only if $V - C$ is an independent set. □

Application of INDEPENDENT SET: NODE COVER

Definition

A node cover of an undirected graph $G = (V, E)$ is a set $C \subseteq V$ that contains at least one endpoint of every edge.

Definition

NODE COVER problem: Given a graph and an integer K, whether there is a node cover C with K or fewer vertices?

Corollary

NODE COVER *is* **NP***-complete.*

Proof.

Outline of proof:
Vertex subset C is a node cover of a graph G if and only if $V - C$ is an independent set. □

Application of INDEPENDENT SET: NODE COVER

Definition

A node cover of an undirected graph $G = (V, E)$ is a set $C \subseteq V$ that contains at least one endpoint of every edge.

Definition

NODE COVER problem: Given a graph and an integer K, whether there is a node cover C with K or fewer vertices?

Corollary

NODE COVER *is* **NP**-*complete.*

Proof.

Outline of proof:
Vertex subset C is a node cover of a graph G if and only if $V - C$ is an independent set. □

Application of INDEPENDENT SET: NODE COVER

Definition

A node cover of an undirected graph $G = (V, E)$ is a set $C \subseteq V$ that contains at least one endpoint of every edge.

Definition

NODE COVER problem: Given a graph and an integer K, whether there is a node cover C with K or fewer vertices?

Corollary

NODE COVER *is* **NP***-complete.*

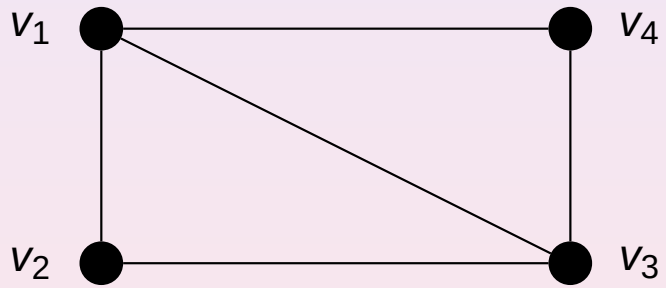
Proof.

Outline of proof:
Vertex subset C is a node cover of a graph G if and only if $V - C$ is an independent set. □

NODE COVER (contd.)

Example

$C = \{v_1, v_3\}$ is a node cover in the graph, $V - C = \{v_2, v_4\}$ is an independent set.



Outline

cut

Definition

A cut in an undirected graph $G = (V, E)$ is a partition of vertices into two non-empty sets S and $V - S$. And the size of a cut $(S, V - S)$ is the number of edges between S and $V - S$.

Definition

MIN-CUT problem: To find a cut with the smallest size in a graph.
MAX-CUT problem: To find a cut with the largest size in a graph.

Observation

MIN-CUT is in **P**.

cut

Definition

A cut in an undirected graph $G = (V, E)$ is a partition of vertices into two non-empty sets S and $V - S$. And the size of a cut $(S, V - S)$ is the number of edges between S and $V - S$.

Definition

MIN-CUT problem: To find a cut with the smallest size in a graph.
MAX-CUT problem: To find a cut with the largest size in a graph.

Observation

MIN-CUT is in **P**.

cut

Definition

A cut in an undirected graph $G = (V, E)$ is a partition of vertices into two non-empty sets S and $V - S$. And the size of a cut $(S, V - S)$ is the number of edges between S and $V - S$.

Definition

MIN-CUT problem: To find a cut with the smallest size in a graph.
MAX-CUT problem: To find a cut with the largest size in a graph.

Observation

MIN-CUT is in **P**.

MAX-CUT

Theorem

MAX-CUT *is* **NP***-complete.*

Proof.

We reduce NAE3SAT to MAX-CUT.
Given m clauses with three literals each, $C_1, C_2, \ldots, C_m$, and the variables are $x_1, x_2, \ldots, x_n$. Then we construct a graph G:
Vertex set $\{x_1, x_2, \ldots, x_n\}$;
Edge: Each clause C_i corresponds to a triangle in G; n_i multiple edges between x_i and $\neg x_i$, where n_i is the number of occurrences of x_i or $\neg x_i$.

□

MAX-CUT

Theorem

MAX-CUT *is* **NP***-complete.*

Proof.

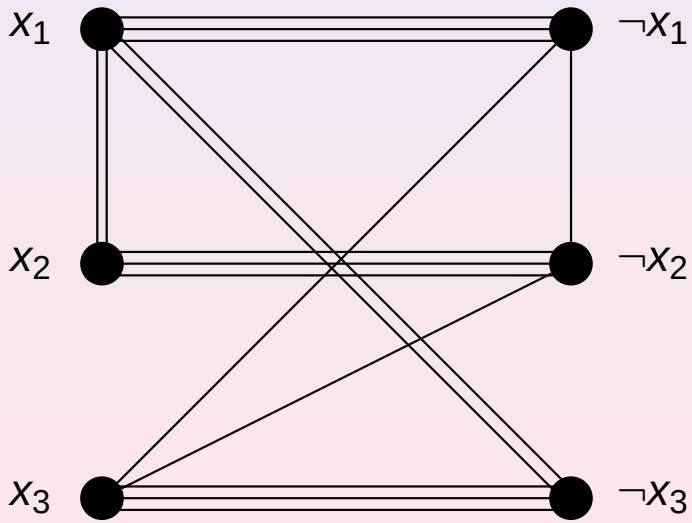
We reduce NAE3SAT to MAX-CUT.
Given m clauses with three literals each, $C_1, C_2, \ldots, C_m$, and the variables are $x_1, x_2, \ldots, x_n$. Then we construct a graph G:
Vertex set $\{x_1, x_2, \ldots, x_n\}$;
Edge: Each clause C_i corresponds to a triangle in G; n_i multiple edges between x_i and $\neg x_i$, where n_i is the number of occurrences of x_i or $\neg x_i$.

□

Proof (contd.)

Example

$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \equiv$
$(x_1 \vee x_2 \vee x_2) \wedge (x_1 \vee \neg x_3 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$



Proof (contd.)

Proof.

Let $K = 5m$. We claim that there is an assignment NAE-satisfying m clauses if and only if there is a cut $(S, V - S)$ with at least K edges in the graph. To see this, if there is an assignment NAE-satisfying all clauses, it is easy to get a cut of size $5m$ (**true** literals form a set S). Conversely, if there is a such a cut, then set the literals in S **true** and literals in $V - S$ **false** and we can get a NAE-satisfying assignment.

□

Proof (contd.)

Proof.

Let $K = 5m$. We claim that there is an assignment NAE-satisfying m clauses if and only if there is a cut $(S, V - S)$ with at least K edges in the graph. To see this, if there is an assignment NAE-satisfying all clauses, it is easy to get a cut of size $5m$ (**true** literals form a set S). Conversely, if there is a such a cut, then set the literals in S **true** and literals in $V - S$ **false** and we can get a NAE-satisfying assignment.

□

Proof (contd.)

Proof.

Let $K = 5m$. We claim that there is an assignment NAE-satisfying m clauses if and only if there is a cut $(S, V - S)$ with at least K edges in the graph. To see this, if there is an assignment NAE-satisfying all clauses, it is easy to get a cut of size $5m$ (**true** literals form a set S). Conversely, if there is a such a cut, then set the literals in S **true** and literals in $V - S$ **false** and we can get a NAE-satisfying assignment.

□