NP-completeness - Part I

K. Subramani\textsuperscript{1}

\textsuperscript{1}Lane Department of Computer Science and Electrical Engineering
West Virginia University

March 30, 2015
1. **NP and NP-completeness**
Outline

1. NP and NP-completeness
2. Boolean Circuits
Outline

1. **NP and NP-completeness**

2. Boolean Circuits

3. The first **NP-complete** problem
1 NP and NP-completeness

2 Boolean Circuits

3 The first NP-complete problem

4 Satisfiability Problems
Certificate definition of NP

NP is the class of problems \( A \) of the following form:

\[ x \text{ is a yes-instance of } A \iff \exists w, (x, w) \text{ is a yes-instance of } B, \]

where \( B \) is a decision problem in \( P \) regarding pairs \((x, w)\) and \(|w| = \text{poly}(|x|)\).

\( w \) is a witness of the fact that \( x \) is a yes-instance. It is called a certificate.

\( w \) is polynomially balanced.
Certificate definition of NP

**Definition**

NP is the class of problems $A$ of the following form: $x$ is a yes-instance of $A$ if and only if there exists a $w$, such that $(x, w)$ is a yes-instance of $B$, where $B$ is a decision problem in P regarding pairs $(x, w)$ and $|w| = \text{poly}(|x|)$.

$w$ is a witness of the fact that $x$ is a yes-instance. It is called a certificate.

NP-completeness

Computational Complexity
Certificate definition of NP

Definition

NP is the class of problems $A$ of the following form:

$x$ is a yes-instance of $A$ if and only if there exists a $w$, such that $(x, w)$ is a yes-instance of $B$, where $B$ is a decision problem in $P$ regarding pairs $(x, w)$ and $|w| = \text{poly}(|x|)$. $w$ is a witness of the fact that $x$ is a yes-instance. It is called a certificate. $w$ is polynomially balanced.
Certificate definition of **NP**

**Definition**

**NP** is the class of problems $A$ of the following form:

$x$ is a yes-instance of $A$ if and only if there exists a $w$, such that $(x, w)$ is a yes-instance of $B$, where $B$ is a decision problem in $P$ regarding pairs $(x, w)$ and $|w| = \text{poly}(|x|)$. $w$ is a witness of the fact that $x$ is a yes-instance. $w$ is polynomially balanced.
Certificate definition of **NP**

**Definition**

**NP** is the class of problems $A$ of the following form:

$x$ is a yes-instance of $A$ if and only if there exists a $w$, such that $(x, w)$ is a yes-instance of $B$, where $B$ is a decision problem in $P$ regarding pairs $(x, w)$ and $|w| = \text{poly}(|x|)$. $w$ is a witness of the fact that $x$ is a yes-instance. It is called a certificate.
Certificate definition of **NP**

**Definition**

**NP** is the class of problems $A$ of the following form:

$x$ is a yes-instance of $A$ if and only if there exists a $w$, such that $(x, w)$ is a yes-instance of $B$,

where $B$ is a decision problem in **P** regarding pairs $(x, w)$ and $|w| = poly(|x|)$. 
Certificate definition of **NP**

**Definition**

**NP** is the class of problems $A$ of the following form:

- $x$ is a yes-instance of $A$ if and only if there exists a $w$, such that $(x, w)$ is a yes-instance of $B$,

where $B$ is a decision problem in $\mathbf{P}$ regarding pairs $(x, w)$ and $|w| = \text{poly}(|x|)$.

$w$ is a witness of the fact that $x$ is a yes-instance.
Certificate definition of \textbf{NP}

\textbf{Definition}

NP is the class of problems \(A\) of the following form:

\[ x \text{ is a yes-instance of } A \text{ if and only if there exists a } w, \text{ such that } (x, w) \text{ is a yes-instance of } B, \]

where \(B\) is a decision problem in \(\mathbf{P}\) regarding pairs \((x, w)\) and \(|w| = poly(|x|)\).

\(w\) is a witness of the fact that \(x\) is a yes-instance. It is called a \textit{certificate}.
Certificate definition of **NP**

**Definition**

**NP** is the class of problems $A$ of the following form:

$x$ is a yes-instance of $A$ if and only if there exists a $w$, such that $(x, w)$ is is a yes-instance of $B$,

where $B$ is a decision problem in $\mathbf{P}$ regarding pairs $(x, w)$ and $|w| = poly(|x|)$.

$w$ is a witness of the fact that $x$ is a yes-instance. It is called a *certificate*.

$w$ is polynomially balanced.
NP and NP-completeness
Boolean Circuits
The first NP-complete problem
Satisfiability Problems

Nondeterministic computation and **NP**

**Definition**

**NP** is the class of problems for which a nondeterministic program exists that runs in time \( \text{poly}(n) \), on instances of length \( n \), such that the input is a yes-instance if and only if there exists a computation path that returns "yes."
Definition

NP is the class of problems for which a nondeterministic program exists that runs in time \( \text{poly}(n) \), on instances of length \( n \), such that the input is a yes-instance if and only if there exists a computation path that returns "yes."
Definition

NP is the class of problems for which a nondeterministic program exists that runs in time $\text{poly}(n)$, on instances of length $n$. 
NP is the class of problems for which a nondeterministic program exists that runs in time $\text{poly}(n)$, on instances of length $n$, such that the input is a yes-instance if and only if there exists a computation path that returns “yes.”
NP and NP-completeness

Boolean Circuits

The first NP-complete problem

Satisfiability Problems

Reductions

Definition

A language \( L_1 \) is reducible to a language \( L_2 \) if there is a function \( R \) from strings of \( L_1 \) to strings of \( L_2 \), such that
\[
(\forall x \in \Sigma^*) \quad x \in L_1 \iff R(x) \in L_2.
\]
Furthermore, the function should be appropriately circumscribed (log space, polynomial time, etc.)

Definition

A polynomial-time reduction is a method of solving one problem by means of a hypothetical subroutine for solving a different problem, that uses polynomial time excluding the time within the subroutine.
Reductions

Definition

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings of $L_2$, such that

\[ (\forall x \in \Sigma^*) \ x \in L_1 \iff R(x) \in L_2. \]

Furthermore, the function should be appropriately circumscribed (log space, polynomial time, etc.).

Definition

A polynomial-time reduction is a method of solving one problem by means of a hypothetical subroutine for solving a different problem, that uses polynomial time excluding the time within the subroutine.
**Reductions**

**Definition**

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings of $L_2$, such that

$$(\forall x \in \Sigma_1^*) \; x \in L_1 \iff R(x) \in L_2.$$
A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings of $L_2$, such that

$$(\forall x \in \Sigma_1^*) \ x \in L_1 \leftrightarrow R(x) \in L_2.$$ 

Furthermore, the function should be appropriately circumscribed.
Definitions

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings of $L_2$, such that

$$(\forall x \in \Sigma_1^*) \ x \in L_1 \iff R(x) \in L_2.$$ 

Furthermore, the function should be appropriately circumscribed (log space,
NP and NP-completeness
Boolean Circuits
The first NP-complete problem
Satisfiability Problems

Reductions

Definition

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings of $L_2$, such that

$$(\forall x \in \Sigma_1^*) \ x \in L_1 \leftrightarrow R(x) \in L_2.$$  

Furthermore, the function should be appropriately circumscribed (log space, polynomial time, etc.)
Reductions

Definition

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings of $L_2$, such that

$$(\forall x \in \Sigma_1^*) \ x \in L_1 \leftrightarrow R(x) \in L_2.$$ 

Furthermore, the function should be appropriately circumscribed (log space, polynomial time, etc.)

Definition
Reductions

**Definition**

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings of $L_2$, such that

$$\forall x \in \Sigma_1^* \quad x \in L_1 \iff R(x) \in L_2.$$ 

Furthermore, the function should be appropriately circumscribed (log space, polynomial time, etc.)

**Definition**

A polynomial-time reduction is a method of solving one problem by means of a hypothetical subroutine for solving a different problem, that uses polynomial time excluding the time within the subroutine.
**Reductions**

**Definition**

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings of $L_2$, such that

$$(\forall x \in \Sigma_1^*) \ x \in L_1 \leftrightarrow R(x) \in L_2.$$ 

Furthermore, the function should be appropriately circumscribed (log space, polynomial time, etc.)

**Definition**

A polynomial-time reduction is a method of solving one problem by means of a hypothetical subroutine for solving a different problem,
Definitions

A language \( L_1 \) is reducible to a language \( L_2 \) if there is a function \( R \) from strings of \( L_1 \) to strings of \( L_2 \), such that

\[
(\forall x \in \Sigma_1^*) \ x \in L_1 \iff R(x) \in L_2.
\]

Furthermore, the function should be appropriately circumscribed (log space, polynomial time, etc.)

A polynomial-time reduction is a method of solving one problem by means of a hypothetical subroutine for solving a different problem, that uses polynomial time excluding the time within the subroutine.
Definition

A polynomial-time many-one reduction (also called Karp reduction) from a problem $A$ to a problem $B$ (both of which are usually required to be decision problems) is a polynomial-time algorithm for transforming inputs to problem $A$ into inputs to problem $B$, such that the transformed problem has the same output as the original problem.

Observation

1. An instance $x$ of problem $A$ can be solved by applying this transformation to produce an instance $y$ of problem $B$, giving $y$ as the input to an algorithm for problem $B$, and returning its output.

2. Polynomial-time many-one reductions are also be known as polynomial transformations or Karp reductions, named after Richard Karp. A reduction of this type may be denoted by the expression $A \leq_{P} B$. 

Karp Reductions
A polynomial-time many-one reduction (also called Karp reduction) from a problem \( A \) to a problem \( B \) (both of which are usually required to be decision problems) is a polynomial-time algorithm for transforming inputs to problem \( A \) into inputs to problem \( B \), such that the transformed problem has the same output as the original problem.

Observation 1: An instance \( x \) of problem \( A \) can be solved by applying this transformation to produce an instance \( y \) of problem \( B \), giving \( y \) as the input to an algorithm for problem \( B \), and returning its output.

Observation 2: Polynomial-time many-one reductions are also known as polynomial transformations or Karp reductions, named after Richard Karp. A reduction of this type may be denoted by the expression \( A \leq_{P} B \).
Definition

A polynomial-time many-one reduction (also called Karp reduction) from a problem $A$ to a problem $B$ (both of which are usually required to be decision problems)
Definition

A polynomial-time many-one reduction (also called Karp reduction) from a problem $A$ to a problem $B$ (both of which are usually required to be decision problems) is a polynomial-time algorithm for transforming inputs to problem $A$ into inputs to problem $B$, such that the transformed problem has the same output as the original problem.

Observation 1

An instance $x$ of problem $A$ can be solved by applying this transformation to produce an instance $y$ of problem $B$, giving $y$ as the input to an algorithm for problem $B$, and returning its output.

Observation 2

Polynomial-time many-one reductions are also be known as polynomial transformations or Karp reductions, named after Richard Karp. A reduction of this type may be denoted by the expression $A \leq_P B$. 
### Definition

A polynomial-time many-one reduction (also called Karp reduction) from a problem $A$ to a problem $B$ (both of which are usually required to be decision problems) is a polynomial-time algorithm for transforming inputs to problem $A$ into inputs to problem $B$, such that the transformed problem has the same output as the original problem.
Karp Reductions

Definition
A polynomial-time many-one reduction (also called Karp reduction) from a problem $A$ to a problem $B$ (both of which are usually required to be decision problems) is a polynomial-time algorithm for transforming inputs to problem $A$ into inputs to problem $B$, such that the transformed problem has the same output as the original problem.

Observation

Observation 1
An instance $x$ of problem $A$ can be solved by applying this transformation to produce an instance $y$ of problem $B$, giving $y$ as the input to an algorithm for problem $B$, and returning its output.

Observation 2
Polynomial-time many-one reductions are also be known as polynomial transformations or Karp reductions, named after Richard Karp. A reduction of this type may be denoted by the expression $A \leq_P B$. 

NP-completeness
Computational Complexity
Definition

A polynomial-time many-one reduction (also called Karp reduction) from a problem $A$ to a problem $B$ (both of which are usually required to be decision problems) is a polynomial-time algorithm for transforming inputs to problem $A$ into inputs to problem $B$, such that the transformed problem has the same output as the original problem.

Observation

1. An instance $x$ of problem $A$ can be solved by applying this transformation to produce an instance $y$ of problem $B$, giving $y$ as the input to an algorithm for problem $B$, and returning its output.
Definition
A polynomial-time many-one reduction (also called Karp reduction) from a problem $A$ to a problem $B$ (both of which are usually required to be decision problems) is a polynomial-time algorithm for transforming inputs to problem $A$ into inputs to problem $B$, such that the transformed problem has the same output as the original problem.

Observation
1. An instance $x$ of problem $A$ can be solved by applying this transformation to produce an instance $y$ of problem $B$, giving $y$ as the input to an algorithm for problem $B$, and returning its output.
2. Polynomial-time many-one reductions are also be known as polynomial transformations or Karp reductions, named after Richard Karp. A reduction of this type may be denoted by the expression $A \leq^P_m B$. 
Turing Reductions

Definition
A polynomial-time Turing reduction from a problem \( A \) to a problem \( B \) is an algorithm that solves problem \( A \) using a polynomial number of calls to a subroutine for problem \( B \), and polynomial time outside of those subroutine calls.

Observation
Polynomial-time Turing reductions are also known as Cook reductions, named after Stephen Cook. A reduction of this type may be denoted by the expression \( A \leq_P^T B \).
Turing Reductions

Definition

A polynomial-time Turing reduction from a problem \( A \) to a problem \( B \) is an algorithm that solves problem \( A \) using a polynomial number of calls to a subroutine for problem \( B \), and polynomial time outside of those subroutine calls.

Observation

Polynomial-time Turing reductions are also known as Cook reductions, named after Stephen Cook. A reduction of this type may be denoted by the expression \( A \leq_P B \).
Definition

A polynomial-time Turing reduction from a problem $A$ to a problem $B$ is an algorithm that solves problem $A$ using a polynomial number of calls to a subroutine for problem $B$, and polynomial time outside of those subroutine calls.

Observation

Polynomial-time Turing reductions are also known as Cook reductions, named after Stephen Cook.
Turing Reductions

**Definition**

A polynomial-time Turing reduction from a problem $A$ to a problem $B$ is an algorithm that solves problem $A$ using a polynomial number of calls to a subroutine for problem $B$, and polynomial time outside of those subroutine calls.

**Observation**

Polynomial-time Turing reductions are also known as Cook reductions, named after Stephen Cook. A reduction of this type may be denoted by the expression $A \leq_P B$. 

NP-completeness

Computational Complexity
Turing Reductions

**Definition**
A polynomial-time Turing reduction from a problem $A$ to a problem $B$ is an algorithm that solves problem $A$ using a polynomial number of calls to a subroutine for problem $B$, and polynomial time outside of those subroutine calls.

**Observation**
*Polynomial-time Turing reductions are also known as Cook reductions, named after Stephen Cook.*
Turing Reductions

Definition

A polynomial-time Turing reduction from a problem $A$ to a problem $B$ is an algorithm that solves problem $A$ using a polynomial number of calls to a subroutine for problem $B$, and polynomial time outside of those subroutine calls.

Observation

Polynomial-time Turing reductions are also known as Cook reductions, named after Stephen Cook.

A reduction of this type may be denoted by the expression $A \leq^P_T B$. 
Transitivity of Reductions

Proposition

If \( A \leq B \) and \( B \leq C \), then \( A \leq C \).

It is understood that both reductions are of the same type.
Transitivity of Reductions

**Proposition**

If $A \leq B$ and $B \leq C$, then $A \leq C$.

It is understood that both reductions are of the same type.
Transitivity of Reductions

**Proposition**

If $A \leq B$
Proposition

If $A \leq B$ and $B \leq C$, then $A \leq C$.

It is understood that both reductions are of the same type.
Transitivity of Reductions

**Proposition**

If $A \leq B$ and $B \leq C$, then $A \leq C$. 
Transitivity of Reductions

**Proposition**

If \( A \leq B \) and \( B \leq C \), then \( A \leq C \). It is understood that both reductions are of the same type.
NP and NP-completeness

Boolean Circuits
The first NP-complete problem
Satisfiability Problems

NP-completeness

Definition

A problem $A$ is said to be NP-complete, if

1. $A \in \text{NP}$.
2. $\forall B \in \text{NP}, B \leq A$.

Observations

1. If only the second condition is satisfied, then the problem is said to be NP-hard.

2. The reductions in question can be Karp or Turing, but we will use Karp for the rest of this chapter.
NP-completeness

Definition

A problem $A$ is said to be NP-complete, if

1. $A \in \text{NP}$.
2. $\forall B \in \text{NP}, B \leq A$.

Observations

1. If only the second condition is satisfied, then the problem is said to be NP-hard.
2. The reductions in question can be Karp or Turing, but we will use Karp for the rest of this chapter.
NP-completeness

Definition

A problem \( A \) is said to be **NP-complete**, if

\[
1. \ A \in \text{NP}.
2. \ \forall \ B \in \text{NP}, \ B \leq A.
\]

Observations

1. If only the second condition is satisfied, then the problem is said to be **NP-hard**.
2. The reductions in question can be Karp or Turing, but we will use Karp for the rest of this chapter.
A problem $A$ is said to be **NP-complete**, if

1. $A \in \text{NP}$.
2. For all $B \in \text{NP}$, $B \leq A$.

Observations:
1. If only the second condition is satisfied, then the problem is said to be **NP-hard**.
2. The reductions in question can be Karp or Turing, but we will use Karp for the rest of this chapter.
Definition

A problem $A$ is said to be **NP-complete**, if

1. $A \in \text{NP}$. 

**Observations**

1. If only the second condition is satisfied, then the problem is said to be **NP-hard**.

2. The reductions in question can be Karp or Turing, but we will use Karp for the rest of this chapter.
A problem $A$ is said to be **NP-complete**, if

1. $A \in \text{NP}$.
2. $\forall B \in \text{NP},$
Definition

A problem $A$ is said to be **NP-complete**, if

1. $A \in \text{NP}$.
2. $\forall B \in \text{NP}, B \leq A$. 
NP-completeness

Definition

A problem $A$ is said to be **NP-complete**, if

1. $A \in \text{NP}$.
2. $\forall B \in \text{NP}, B \leq A$.

Observations

- If only the second condition is satisfied, then the problem is said to be **NP-hard**.
- The reductions in question can be Karp or Turing, but we will use Karp for the rest of this chapter.
NP-completeness

Definition

A problem $A$ is said to be **NP-complete**, if

1. $A \in \text{NP}$.
2. $\forall B \in \text{NP}, B \leq A$.

Observations

1. If only the second condition is satisfied,
NP-completeness

Definition

A problem \( A \) is said to be \textbf{NP-complete}, if

1. \( A \in \text{NP} \).
2. \( \forall B \in \text{NP}, B \leq A \).

Observations

1. If only the second condition is satisfied, then the problem is said to be \textbf{NP-hard}.
### Definition

A problem $A$ is said to be **NP-complete**, if

1. $A \in \text{NP}$.  
2. $\forall B \in \text{NP}, B \leq A$.  

### Observations

1. *If only the second condition is satisfied, then the problem is said to be NP-hard.*  
2. *The reductions in question can be Karp or Turing, but we will use Karp for the rest of this chapter.*
Boolean Circuits (Syntax)

A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.

The nodes $V = \{1, 2, ..., n\}$ are called the gates of $C$.

We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$.

Each gate $i$ has a sort $s(i)$ associated with it, where $s(i) \in \{true, false\} \cup \{x_1, x_2, ..., x_n\} \cup \{\lor, \land, \neg\}$.

If $s(i) \in \{true, false\} \cup \{x_1, x_2, ..., x_n\}$, then its in-degree is 0.

If $s(i) \in \{\neg\}$, its in-degree is 1.

All other gates have in-degree 2.

All gates except gate $n$ have out-degree 1.

Gate $n$, is called the output gate and has out-degree 0.
A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.

The nodes $V = \{1, 2, ..., n\}$ are called the gates of $C$.

We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$.

Each gate $i$ has a sort $s(i)$ associated with it, where $s(i) \in \{true, false\} \cup \{x_1, x_2, ..., x\} \cup \{\lor, \land, \lnot\}$.

If $s(i) \in \{true, false\} \cup \{x_1, x_2, ..., x\}$, then its in-degree is 0.

If $s(i) \in \{\lnot\}$, its in-degree is 1.

All other gates have in-degree 2.

All gates except gate $n$ have out-degree 1.

Gate $n$, is called the output gate and has out-degree 0.
A boolean circuit \( C \) is a DAG \( G = \langle V, E \rangle \).
Boolean Circuits (Syntax)

Syntax

1. A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of $C$. 
NP and NP-completeness

Boolean Circuits

The first NP-complete problem

Satisfiability Problems

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots, n\}$ are called the gates of $C$.
3. We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$. 
Boolean Circuits (Syntax)

1. A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots, n\}$ are called the gates of $C$.
3. We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$.
4. Each gate $i$ has a sort $s(i)$ associated with it, where $s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\} \cup \{\lor, \land, \neg\}$.
A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.

The nodes $V = \{1, 2, \ldots n\}$ are called the gates of $C$.

We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$.

Each gate $i$ has a sort $s(i)$ associated with it, where
$s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\} \cup \{\lor, \land, \neg\}$.

If $s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
Boolean Circuits (Syntax)

Syntax

1. A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots, n\}$ are called the gates of $C$.
3. We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$.
4. Each gate $i$ has a sort $s(i)$ associated with it, where $s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\} \cup \{\lor, \land, \neg\}$.
5. If $s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
6. If $s(i) \in \{\neg\}$, its in-degree is 1.
Boolean Circuits (Syntax)

1. A boolean circuit $C$ is a DAG $G = (V, E)$.
2. The nodes $V = \{1, 2, \ldots, n\}$ are called the gates of $C$.
3. We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$.
4. Each gate $i$ has a sort $s(i)$ associated with it, where $s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\} \cup \{\lor, \land, \neg\}$.
5. If $s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
6. If $s(i) \in \{\neg\}$, its in-degree is 1.
7. All other gates have in-degree 2.
Boolean Circuits (Syntax)

1. A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots, n\}$ are called the gates of $C$.
3. We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$.
4. Each gate $i$ has a sort $s(i)$ associated with it, where $s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\} \cup \{\lor, \land, \neg\}$.
5. If $s(i) \in \{\text{true, false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
6. If $s(i) \in \{\neg\}$, its in-degree is 1.
7. All other gates have in-degree 2.
8. All gates except gate $n$ have out-degree 1.
Boolean Circuits (Syntax)

1. A boolean circuit $C$ is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots, n\}$ are called the gates of $C$.
3. We can assume without loss of generality that the edges are of the form $(i, j)$, where $i < j$.
4. Each gate $i$ has a sort $s(i)$ associated with it, where $s(i) \in \{\text{true}, \text{false}\} \cup \{x_1, x_2, \ldots\} \cup \{\lor, \land, \neg\}$.
5. If $s(i) \in \{\text{true}, \text{false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
6. If $s(i) \in \{\neg\}$, its in-degree is 1.
7. All other gates have in-degree 2.
8. All gates except gate $n$ have out-degree 1.
9. Gate $n$, is called the output gate and has out-degree 0.
Boolean Circuits (Semantics)

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment. This value can be computed inductively as follows:

1. If the gate is \textit{true} or \textit{false}, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort \(\neg\), then its value is the complement of its input.
4. If the gate has sort \(\lor\), then its value is \textit{true} if at least one of its two input gates has value \textit{true} and is \textit{false} otherwise.
5. If the gate has sort \(\land\), then its value is \textit{true} if both its two input gates have value \textit{true} and is \textit{false} otherwise.
6. The value of the circuit is the value of the output gate.
The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment. This value can be computed inductively as follows:

1. If the gate is true or false, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort \( \neg \), then its value is the complement of its input.
4. If the gate has sort \( \lor \), then its value is true if at least one of its two input gates has value true and is false otherwise.
5. If the gate has sort \( \land \), then its value is true if both its two input gates have value true and is false otherwise.
6. The value of the circuit is the value of the output gate.
The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.
Boolean Circuits (Semantics)

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is `true` or `false`, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort `¬`, then its value is the complement of its input.
4. If the gate has sort `∨`, then its value is `true` if at least one of its two input gates has value `true` and is `false` otherwise.
5. If the gate has sort `∧`, then its value is `true` if both its two input gates have value `true` and is `false` otherwise.
6. The value of the circuit is the value of the output gate.
Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is true or false, then it retains that value.
Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is **true** or **false**, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is true or false, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort \( \neg \), then its value is the complement of its input.
4. If the gate has sort \( \lor \), then its value is true if at least one of its two input gates has value true and is false otherwise.
5. If the gate has sort \( \land \), then its value is true if both its two input gates have value true and is false otherwise.
6. The value of the circuit is the value of the output gate.
The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is true or false, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort \( \neg \), then its value is the complement of its input.
4. If the gate has sort \( \lor \), then its value is true if at least one of its two input gates has value true and is false otherwise.
The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is true or false, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort $\neg$, then its value is the complement of its input.
4. If the gate has sort $\lor$, then its value is true if at least one of its two input gates has value true and is false otherwise.
5. If the gate has sort $\land$, then its value is true if both its two input gates have value true and is false otherwise.
Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is true or false, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort ¬, then its value is the complement of its input.
4. If the gate has sort ∨, then its value is true if at least one of its two input gates has value true and is false otherwise.
5. If the gate has sort ∧, then its value is true if both its two input gates have value true and is false otherwise.
6. The value of the circuit is the value of the output gate.
Size and Depth

Definition
The size of a boolean circuit is the number of gates in that circuit.

Definition
The depth of a circuit is the maximum distance from an input gate to the output gate.
Size and Depth

Definition

The size of a boolean circuit is the number of gates in that circuit.

Definition

The depth of a circuit is the maximum distance from an input gate to the output gate.
Size and Depth

**Definition**
The size of a boolean circuit is the number of gates in that circuit.
Size and Depth

**Definition**

The size of a boolean circuit is the number of gates in that circuit.
Size and Depth

Definition
The size of a boolean circuit is the number of gates in that circuit.

Definition
The depth of a circuit is the maximum distance from an input gate to the output gate.
### String acceptance

Consider an $n$-input boolean circuit. We say that a string $x$, with $|x| = n$ and $x_i \in \{0, 1\}$ is accepted by a circuit, if the output of the circuit is `true` when presented with this string. The $i$th input is `true` if and only if $x_i = 1$. 
String acceptance

Definition

Consider an $n$-input boolean circuit. We say that a string $x$, with $|x| = n$ and $x_i \in \{0, 1\}$, is accepted by a circuit, if the output of the circuit is true, when presented with this string. The $i$th input is true if and only if $x_i = 1$. 
String acceptance

Definition

Consider an $n$-input boolean circuit.
Definition

Consider an $n$-input boolean circuit. We say that a string $x$, with $|x| = n$ and $x_i \in \{0, 1\}$ is accepted by a circuit,
String acceptance

Definition

Consider an $n$-input boolean circuit. We say that a string $x$, with $|x| = n$ and $x_i \in \{0, 1\}$ is accepted by a circuit, if the output of the circuit is true, when presented with this string.
String acceptance

**Definition**

Consider an $n$-input boolean circuit. We say that a string $x$, with $|x| = n$ and $x_i \in \{0, 1\}$ is accepted by a circuit, if the output of the circuit is **true**, when presented with this string.

The $i^{th}$ input is **true** if and only if $x_i = 1$. 

---

**Boolean Circuits**

The first NP-complete problem

**Satisfiability Problems**
Language Acceptance

Observations

1. The above definition holds only for fixed $n$.

2. We can generalize the definition to strings of arbitrary length.
Observations

1. The above definition holds only for fixed $n$.
2. We can generalize the definition to strings of arbitrary length.
Observations

1. The above definition holds only for fixed $n$. 
Language Acceptance

Observations

1. The above definition holds only for fixed $n$.
2. We can generalize the definition to strings of arbitrary length.
Circuit Families

Definition

A family of circuits is an infinite sequence $C = (C_0, C_1, ...)$ of Boolean circuits, where $C_n$ has $n$ input variables.

Definition

A language $L \subseteq \{0, 1\}^*$ has polynomial circuits, if there is a family of circuits $C = (C_0, C_1, ...)$ such that:

1. The size of $C_n$ is at most $p(n)$, for some fixed polynomial $p(n)$.
2. $\forall x \in \{0, 1\}^*$, $x \in L$ if and only if, the output of $C|_x$ is true, when the $i$th input variable is true if $x_i = 1$ and false otherwise.
Circuit Families

Definition

A family of circuits is an infinite sequence \( C = (C_0, C_1, \ldots) \) of Boolean circuits, where \( C_n \) has \( n \) input variables.

A language \( L \subseteq \{0, 1\}^* \) has polynomial circuits, if there is a family of circuits \( C = (C_0, C_1, \ldots) \) such that:

1. The size of \( C_n \) is at most \( p(n) \), for some fixed polynomial \( p(n) \).
2. \( \forall x \in \{0, 1\}^* \), \( x \in L \) if and only if, the output of \( C_x \) is true, when the \( i \)th input variable is true if \( x_i = 1 \) and false otherwise.
Circuit Families

Definition

A family of circuits is an infinite sequence $C = (C_0, C_1, \ldots)$ of Boolean circuits, where $C_n$ has $n$ input variables.
A family of circuits is an infinite sequence $C = (C_0, C_1, \ldots)$ of Boolean circuits, where $C_n$ has $n$ input variables.
### Circuit Families

**Definition**

A family of circuits is an infinite sequence $C = (C_0, C_1, \ldots)$ of Boolean circuits, where $C_n$ has $n$ input variables.

**Definition**

A language $L \subseteq \{0, 1\}^*$ has polynomial circuits, if there is a family of circuits $C = (C_0, C_1, \ldots)$ such that:

1. The size of $C_n$ is at most $p(n)$, for some fixed polynomial $p(n)$.
2. $\forall x \in \{0, 1\}^*$, $x \in L$ if and only if, the output of $C|x$ is true, when the $i$th input variable is true if $x_i = 1$ and false otherwise.
Circuit Families

**Definition**

A family of circuits is an infinite sequence $C = (C_0, C_1, \ldots)$ of Boolean circuits, where $C_n$ has $n$ input variables.

**Definition**

A language $L \subseteq \{0, 1\}^*$ has polynomial circuits, if there is a family of circuits $C = (C_0, C_1, \ldots)$ such that:

1. The size of $C_n$ is at most $p(n)$, for some fixed polynomial $p(n)$. 
Circuit Families

### Definition

A family of circuits is an infinite sequence \( C = (C_0, C_1, \ldots) \) of Boolean circuits, where \( C_n \) has \( n \) input variables.

### Definition

A language \( L \subseteq \{0, 1\}^* \) has polynomial circuits, if there is a family of circuits \( C = (C_0, C_1, \ldots) \) such that:

1. The size of \( C_n \) is at most \( p(n) \), for some fixed polynomial \( p(n) \).
2. \( \forall x \in \{0, 1\}^*, x \in L \) if and only if, the output of \( C_{|x|} \) is **true**, when the \( i^{th} \) input variable is **true** if \( x_i = 1 \) and **false** otherwise.
Uniform circuit families

Definition
A family of circuits $C = (C_0, C_1, ...)$ is said to be uniform if there is log-space bounded algorithm which on input $1^n$ outputs $C_n$.

Definition
A language $L$ has uniformly polynomial circuits if there is a uniform family of circuits that decides it.
Uniform circuit families

Definition

A family of circuits $C = (C_0, C_1, ...)$ is said to be uniform if there is a log-space bounded algorithm which on input $1^n$ outputs $C_n$.

Definition

A language $L$ has uniformly polynomial circuits if there is a uniform family of circuits that decides it.
Uniform circuit families

Definition

A family of circuits $C = (C_0, C_1, \ldots)$ is said to be *uniform* if there is log-space bounded algorithm which on input $1^n$ outputs $C_n$. 
Uniform circuit families

**Definition**

A family of circuits $\mathcal{C} = (C_0, C_1, \ldots)$ is said to be *uniform* if there is log-space bounded algorithm which on input $1^n$ outputs $C_n$. 

**Definition**
Uniform circuit families

**Definition**

A family of circuits $C = (C_0, C_1, \ldots)$ is said to be *uniform* if there is log-space bounded algorithm which on input $1^n$ outputs $C_n$.

**Definition**

A language $L$ has uniformly polynomial circuits if there is a uniform family of circuits that decides it.
P and uniform circuit families

Theorem

A language L is in $\mathcal{P}$ if and only if it has uniformly polynomial circuits.

NP-completeness

Computational Complexity
P and uniform circuit families

Theorem

A language L is in P if and only if it has uniformly polynomial circuits.
Theorem

A language $L$ is in $\textbf{P}$ if and only if it has uniformly polynomial circuits.
The first **NP-complete** problem

How many languages are there in NP?

The task of proving a language to be NP-complete is formidable, because we have to show that every language in NP reduces to the language in question.

However, once we have shown a language $L$ to be NP-complete, we can show all other languages to be NP-complete, by reducing $L$ to these languages!

So which language (or problem) is the first NP-complete language (problem)?
The first \textbf{NP-complete} problem

Motivation
Motivation

1. How many languages are there in NP?
The first **NP-complete** problem

**Motivation**

1. How many languages are there in **NP**?
2. The task of proving a language to be **NP-complete** is formidable,
The first NP-complete problem

Motivation

1. How many languages are there in NP?

2. The task of proving a language to be **NP-complete** is formidable, because we have to show that every language in NP reduces to the language in question.
The first **NP-complete** problem

**Motivation**

1. How many languages are there in **NP**?
2. The task of proving a language to be **NP-complete** is formidable, because we have to show that every language in **NP** reduces to the language in question.
3. However, once we have shown a language $L$ to be **NP-complete**, we can show all other languages to be **NP-complete**, by reducing $L$ to these languages!
The first **NP-complete** problem

**Motivation**

1. How many languages are there in **NP**?
2. The task of proving a language to be **NP-complete** is formidable, because we have to show that every language in **NP** reduces to the language in question.
3. However, once we have shown a language $L$ to be **NP-complete**, we can show all other languages to be **NP-complete**, by reducing $L$ to these languages!
4. So which language (or problem) is the first **NP-complete** language (problem)?
CircuitSAT

Theorem

CircuitSAT is NP-complete.

Proof

1. Let A be any language in NP.
2. A must have a polynomial time verifier V, such that x ∈ A if and only if V accepts ⟨x, y⟩ for some polynomially balanced y.
3. Since V runs in polynomial time, we know that there exists a uniform family of polynomial size circuits C that decides the language decided by V; i.e., C is equivalent to V.
4. The input of C is ⟨x, y⟩ and a specific C ∈ C can be constructed in time polynomial in |x| and |y|.
**Theorem**

*CircuitSAT is NP-complete.*
Theorem

*CircuitSAT* is **NP-complete**.

Proof
Theorem

_CircuitSAT is **NP-complete.**_

Proof

CircuitSAT is clearly in **NP.**
Theorem

CircuitSAT is NP-complete.

Proof

CircuitSAT is clearly in NP.
Theorem

*CircuitSAT* is **NP-complete**.

Proof

CircuitSAT is clearly in **NP**.

1. Let $A$ be any language in **NP**.
**Theorem**

*CircuitSAT is NP-complete.*

**Proof**

*CircuitSAT is clearly in NP.*

1. Let $A$ be any language in NP.
2. $A$ must have a polynomial time verifier $V$, such that $x \in A$ if and only if $V$ accepts $\langle x, y \rangle$ for some polynomially balanced $y$. 
**Theorem**

*CircuitSAT is NP-complete.*

**Proof**

CircuitSAT is clearly in **NP**.

1. Let $A$ be any language in **NP**.
2. $A$ must have a polynomial time verifier $V$, such that $x \in A$ if and only if $V$ accepts $\langle x, y \rangle$ for some polynomially balanced $y$.
3. Since $V$ runs in polynomial time, we know that there exists a uniform family of polynomial size circuits $C$ that decides the language decided by $V$;
Theorem

*CircuitSAT is NP-complete.*

Proof

CircuitSAT is clearly in NP.

1. Let $A$ be any language in NP.
2. $A$ must have a polynomial time verifier $V$, such that $x \in A$ if and only if $V$ accepts $\langle x, y \rangle$ for some polynomially balanced $y$.
3. Since $V$ runs in polynomial time, we know that there exists a uniform family of polynomial size circuits $C$ that decides the language decided by $V$; i.e., $C$ is equivalent to $V$. 
CircuitSAT

**Theorem**

*CircuitSAT is **NP-complete**.*

**Proof**

CircuitSAT is clearly in **NP**.

1. Let \( A \) be any language in **NP**.
2. \( A \) must have a polynomial time verifier \( V \), such that \( x \in A \) if and only if \( V \) accepts \( \langle x, y \rangle \) for some polynomially balanced \( y \).
3. Since \( V \) runs in polynomial time, we know that there exists a uniform family of polynomial size circuits \( C \) that decides the language decided by \( V \); i.e., \( C \) is equivalent to \( V \).
4. The input of \( C \) is \( \langle x, y \rangle \) and a specific \( C \in C \) can be constructed in time polynomial in \( |x| \) and \( |y| \).
Completing the reduction

The reduction from $A$ to $C$ is as follows:

1. Given an input $x$, output a description of the circuit $C(x, y)$, with the $x$ values set to the given values and the $y$ values left as variables.

2. The resulting circuit is satisfiable if and only if $x \in A$.

3. The reduction is clearly polynomial time, since $C$ is uniform.
Completing the reduction

Proof (contd.)

The reduction from $A$ to $C$ is as follows:

1. Given an input $x$, output a description of the circuit $C(x, y)$, with the $x$ values set to the given values and the $y$ values left as variables.

2. The resulting circuit is satisfiable if and only if $x \in A$.

3. The reduction is clearly polynomial time, since $C$ is uniform.
Completing the reduction

Proof (contd.)

The reduction from $A$ to $C$ is as follows:

1. Given an input $x$, output a description of the circuit $C(x, y)$, with the $x$ values set to the given values and the $y$ values left as variables.
2. The resulting circuit is satisfiable if and only if $x \in A$.
3. The reduction is clearly polynomial time, since $C$ is uniform.
Proof (contd.)

The reduction from $A$ to $C$ is as follows:

1. Given an input $x$, output a description of the circuit $C(x, y)$, with the $x$ values set to the given values and the $y$ values left as variables.
Completing the reduction

Proof (contd.)

The reduction from $A$ to $C$ is as follows:

1. Given an input $x$, output a description of the circuit $C(x, y)$, with the $x$ values set to the given values and the $y$ values left as variables.

2. The resulting circuit is satisfiable if and only if $x \in A$. 
Completing the reduction

Proof (contd.)

The reduction from $A$ to $C$ is as follows:

1. Given an input $x$, output a description of the circuit $C(x, y)$, with the $x$ values set to the given values and the $y$ values left as variables.
2. The resulting circuit is satisfiable if and only if $x \in A$.
3. The reduction is clearly polynomial time, since $C$ is uniform.
Witness Existence

**Definition**

**Input:** A program \( P(x, w) \), an input \( x \) and an integer \( t \) given in unary.

**Query:** Does there exist a \( w \), with \(|w| \leq t\), such that \( P(x, w) \) returns "yes" after at most \( t \) steps?

**Observations**

1. Why is the \( \text{WITNESS-EXISTENCE} \) problem NP-complete?
2. In the textbook, they reduce \( \text{WITNESS-EXISTENCE} \) to CircuitSAT.
3. In his seminal 1971 paper, Cook reduced the \( \text{WITNESS-EXISTENCE} \) problem directly to SAT.

Witness Existence

Definition

Input: 
A program \( P(x, w) \), an input \( x \) and an integer \( t \) given in unary.

Query: 
Does there exist a \( w \), with \( |w| \leq t \), such that \( P(x, w) \) returns "yes" after at most \( t \) steps?

Observations

1. Why is the \textsc{Witness-Existence} problem NP-complete?
2. In the textbook, they reduce \textsc{Witness-Existence} to CircuitSAT.
3. In his seminal 1971 paper, Cook reduced the \textsc{Witness-Existence} problem directly to SAT.

Definition

**Input:** A program $P(x, w)$, an input $x$ and an integer $t$ given in unary.
Witness Existence

Definition

**Input:** A program $P(x, w)$, an input $x$ and an integer $t$ given in unary.

**Query:** Does there exist a $w$,
Witness Existence

Definition

**Input:** A program $P(x, w)$, an input $x$ and an integer $t$ given in unary.

**Query:** Does there exist a $w$, with $|w| \leq t$, such that $P(x, w)$ returns “yes” after at most $t$ steps?
**Witness Existence**

**Definition**

**Input:** A program $P(x, w)$, an input $x$ and an integer $t$ given in unary.

**Query:** Does there exist a $w$, with $|w| \leq t$, such that $P(x, w)$ returns “yes” after at most $t$ steps?

**Observations**

1. Why is the WITNESS-EXISTENCE problem NP-complete?
2. In the textbook, they reduce WITNESS-EXISTENCE to CircuitSAT.
**Witness Existence**

**Definition**

**Input:** A program $P(x, w)$, an input $x$ and an integer $t$ given in unary.

**Query:** Does there exist a $w$, with $|w| \leq t$, such that $P(x, w)$ returns “yes” after at most $t$ steps?

**Observations**

1. *Why is the Witness-Existence problem NP-complete?*
**Definition**

**Input:** A program $P(x, w)$, an input $x$ and an integer $t$ given in unary.

**Query:** Does there exist a $w$, with $|w| \leq t$, such that $P(x, w)$ returns “yes” after at most $t$ steps?

**Observations**

1. *Why is the Witness-Existence problem NP-complete?*
2. *In the textbook, they reduce Witness-Existence to CircuitSAT.*
Witness Existence

**Definition**

**Input:** A program $P(x, w)$, an input $x$ and an integer $t$ given in unary.

**Query:** Does there exist a $w$, with $|w| \leq t$, such that $P(x, w)$ returns “yes” after at most $t$ steps?

**Observations**

1. *Why is the Witness-Existence problem NP-complete?*
2. *In the textbook, they reduce Witness-Existence to CircuitSAT.*
3. *In his seminal 1971 paper, Cook reduced the Witness-Existence problem directly to SAT.*
**Witness Existence**

**Definition**

**Input:** A program $P(x, w)$, an input $x$ and an integer $t$ given in unary.

**Query:** Does there exist a $w$, with $|w| \leq t$, such that $P(x, w)$ returns “yes” after at most $t$ steps?

**Observations**

1. *Why is the Witness-Existence problem NP-complete?*
2. *In the textbook, they reduce Witness-Existence to CircuitSAT.*
3. *In his seminal 1971 paper, Cook reduced the Witness-Existence problem directly to SAT.*
   
Satisfiability (SAT)

Definition
Input: A boolean formula $\varphi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\varphi = C_1 \land C_2 \ldots C_m$.
Query: Is $\varphi$ satisfiable?

Theorem
SAT is NP-complete.

Proof
SAT is clearly in NP. (Why?)
Clearly, CircuitSAT $\leq$ SAT (Previous chapter).
Satisfiability (SAT)

Definition

Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$.

Query: Is $\phi$ satisfiable?

Theorem SAT is NP-complete.

Proof

SAT is clearly in NP. (Why?)

Clearly, CircuitSAT $\leq$ SAT (Previous chapter).
Satisfiability (SAT)

Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses,
Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,

$$\phi = C_1 \land C_2 \ldots C_m.$$
**Definition**

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,

$$\phi = C_1 \land C_2 \ldots C_m.$$  

**Query:** Is $\phi$ satisfiable?
Satisfiability (SAT)

Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., 
$\phi = C_1 \land C_2 \ldots C_m$.

**Query:** Is $\phi$ satisfiable?

Theorem

SAT is NP-complete.

Proof

SAT is clearly in NP. (Why?)

Clearly, CircuitSAT ≤ SAT (Previous chapter).
Satisfiability (SAT)

**Definition**

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,

$$\phi = C_1 \land C_2 \ldots C_m.$$  

**Query:** Is $\phi$ satisfiable?

**Theorem**

*SAT is NP-complete.*
Satisfiability (SAT)

**Definition**

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$.

**Query:** Is $\phi$ satisfiable?

**Theorem**

*SAT is NP-complete.*

**Proof**

SAT is clearly in NP. (Why?) Clearly, CircuitSAT ≤ SAT (Previous chapter).
Satisfiability (SAT)

**Definition**

**Input:** A boolean formula \( \phi \) in CNF form over \( n \) variables and \( m \) clauses, i.e.,

\[ \phi = C_1 \land C_2 \ldots C_m. \]

**Query:** Is \( \phi \) satisfiable?

**Theorem**

*SAT is NP-complete.*

**Proof**

SAT is clearly in \( \textbf{NP} \).
Satisfiability (SAT)

Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$.

**Query:** Is $\phi$ satisfiable?

Theorem

**SAT is NP-complete.**

Proof

SAT is clearly in **NP**. (Why?)
Definition

Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,
$\phi = C_1 \land C_2 \ldots C_m$.

Query: Is $\phi$ satisfiable?

Theorem

$SAT$ is NP-complete.

Proof

SAT is clearly in NP. (Why?)

Clearly, CircuitSAT $\leq$ SAT
### Satisfiability (SAT)

**Definition**

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,
$$\phi = C_1 \land C_2 \ldots C_m.$$  

**Query:** Is $\phi$ satisfiable?

**Theorem**

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

**Proof**

SAT is clearly in **NP**. (Why?)

Clearly, CircuitSAT $\leq$ SAT (Previous chapter).
3SAT

Satisfiability Problems

3SAT

Definition
Input:
A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e.,

$\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 3 literals.

Query:
Is $\phi$ satisfiable?

Observations
1. 3SAT is clearly in NP.
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3. Consider a clause in 2CNF form. Can you represent it using 3CNF form?
4. Consider a clause in 4CNF form. Can you represent it using 3CNF form?
5. Generalize...
6. 3SAT is the most versatile of SAT problems.

NP and NP-completeness
Boolean Circuits
The first NP-complete problem
Satisfiability Problems

NP-completeness

Computational Complexity
3SAT

Definition

Input: A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 3 literals.

Query: Is $\phi$ satisfiable?

Observations
1. 3SAT is clearly in NP.
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3. Consider a clause in 2CNF form. Can you represent it using 3CNF form?
4. Consider a clause in 4CNF form. Can you represent it using 3CNF form?
5. Generalize...!
6. 3SAT is the most versatile of SAT problems.
3SAT

Definition

**Input:** A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses,
3SAT

Definition

**Input:** A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 3 literals.
**Definition**

**Input:** A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 3 literals.

**Query:** Is $\phi$ satisfiable?
**3SAT**

**Definition**

**Input:** A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e.,
$\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 3 literals.

**Query:** Is $\phi$ satisfiable?

**Observations**

1. 3SAT is clearly in NP.
2. Can you represent a clause in 1CNF form using 3CNF form?
3. Can you represent a clause in 2CNF form using 3CNF form?
4. Can you represent a clause in 4CNF form using 3CNF form?
5. Generalize...
6. 3SAT is the most versatile of SAT problems.
3SAT

Definition

Input: A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e.,

$\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 3 literals.

Query: Is $\phi$ satisfiable?

Observations

1. 3SAT is clearly in NP.
3SAT

Definition

Input: A boolean formula \( \phi \) in 3CNF form over \( n \) variables and \( m \) clauses, i.e., \( \phi = C_1 \land C_2 \ldots C_m \), with each clause having exactly 3 literals.

Query: Is \( \phi \) satisfiable?

Observations

1. 3SAT is clearly in \textbf{NP}.
2. Consider a clause in 1CNF form.
3SAT

Definition

Input: A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e.,
$\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 3 literals.

Query: Is $\phi$ satisfiable?

Observations

1. 3SAT is clearly in NP.
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3SAT

Definition

Input: A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e.,
$$\phi = C_1 \land C_2 \ldots C_m,$$
with each clause having exactly 3 literals.

Query: Is $\phi$ satisfiable?

Observations

1. 3SAT is clearly in NP.
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3. Consider a clause in 2CNF form.
**Definition**

**Input:** A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e.,

$$\phi = C_1 \land C_2 \ldots C_m,$$

with each clause having exactly 3 literals.

**Query:** Is $\phi$ satisfiable?

**Observations**

1. **3SAT is clearly in \textbf{NP}**.
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3. Consider a clause in 2CNF form. Can you represent it using 3CNF form?
Definition

**Input:** A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 3 literals.

**Query:** Is $\phi$ satisfiable?

Observations

1. 3SAT is clearly in NP.
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3. Consider a clause in 2CNF form. Can you represent it using 3CNF form?
4. Consider a clause in 4CNF form.
Definition

Input: A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 3 literals.

Query: Is $\phi$ satisfiable?

Observations

1. **3SAT is clearly in NP.**
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3. Consider a clause in 2CNF form. Can you represent it using 3CNF form?
4. Consider a clause in 4CNF form. Can you represent it using 3CNF form?
3SAT

Definition

Input: A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 3 literals.

Query: Is $\phi$ satisfiable?

Observations

1. 3SAT is clearly in NP.
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3. Consider a clause in 2CNF form. Can you represent it using 3CNF form?
4. Consider a clause in 4CNF form. Can you represent it using 3CNF form?
5. Generalize . . . !
3SAT

Definition

**Input:** A boolean formula $\phi$ in 3CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 3 literals.

**Query:** Is $\phi$ satisfiable?

Observations

1. 3SAT is clearly in NP.
2. Consider a clause in 1CNF form. Can you represent it using 3CNF form?
3. Consider a clause in 2CNF form. Can you represent it using 3CNF form?
4. Consider a clause in 4CNF form. Can you represent it using 3CNF form?
5. Generalize . . .!
6. 3SAT is the most versatile of SAT problems.
NAESAT

Definition

An assignment to a boolean formula is nae-satisfying, if
1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition

Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$.
Query: Is $\phi$ nae-satisfiable?

Reduction

1. Construct a new formula $\phi'$ by adding a new variable $s$ to every single clause.
2. If $\phi$ is satisfiable, then $\phi'$ is nae-satisfiable.
3. If $\phi'$ is nae-satisfiable, then $\phi$ must be satisfiable. (Why?)
4. Thus, SAT $\leq$ NAESAT.
Definition

An assignment to a boolean formula is nae-satisfying, if
1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition

Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots \land C_m$.
Query: Is $\phi$ nae-satisfiable?

Reduction

1. Construct a new formula $\phi'$ by adding a new variable $s$ to every single clause.
2. If $\phi$ is satisfiable, then $\phi'$ is nae-satisfiable.
3. If $\phi'$ is nae-satisfiable, then $\phi$ must be satisfiable. (Why?)
4. Thus, SAT $\leq$ NAESAT.
NAESAT

Definition

An assignment to a boolean formula is nae-satisfying, if

1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition

Input:
A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,

$$\phi = C_1 \land C_2 \ldots C_m.$$ 

Query:
Is $\phi$ nae-satisfiable?

Reduction

1. Construct a new formula $\phi'$ by adding a new variable $s$ to every single clause.
2. If $\phi$ is satisfiable, then $\phi'$ is nae-satisfiable.
3. If $\phi'$ is nae-satisfiable, then $\phi$ must be satisfiable. (Why?)
4. Thus, SAT $\leq$ NAESAT.
Definition

An assignment to a boolean formula is nae-satisfying, if

1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

**Definition**

Input:
A boolean formula \( \phi \) in CNF form over \( n \) variables and \( m \) clauses, i.e., \( \phi = C_1 \land C_2 \land ... \land C_m \).

Query:
Is \( \phi \) nae-satisfiable?

**Reduction**

1. Construct a new formula \( \phi' \) by adding a new variable \( s \) to every single clause.
2. If \( \phi \) is satisfiable, then \( \phi' \) is nae-satisfiable.
3. If \( \phi' \) is nae-satisfiable, then \( \phi \) must be satisfiable. (Why?)
4. Thus, SAT \( \leq \) NAESAT.
NAESAT

Definition

An assignment to a boolean formula is nae-satisfying, if

1. It satisfies at least one literal in each clause.
Definition

An assignment to a boolean formula is nae-satisfying, if

1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.
Definition

An assignment to a boolean formula is nae-satisfying, if

1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition
### Definition

An assignment to a boolean formula is nae-satisfying, if

1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

### Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses,
## NAESAT

### Definition

An assignment to a boolean formula is nae-satisfying, if:

1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

### Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,

$$\phi = C_1 \land C_2 \ldots C_m.$$
NP and NP-completeness
Boolean Circuits
The first NP-complete problem
Satisfiability Problems

NAESAT

**Definition**

An assignment to a boolean formula is nae-satisfying, if

1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

**Definition**

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,
$$\phi = C_1 \land C_2 \ldots C_m.$$  
**Query:** Is $\phi$ nae-satisfiable?
Definition
An assignment to a boolean formula is nae-satisfying, if
1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition
Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,
$\phi = C_1 \land C_2 \ldots C_m$.
Query: Is $\phi$ nae-satisfiable?

Reduction
NAESAT

Definition
An assignment to a boolean formula is nae-satisfying, if
1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition
Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,
$\phi = C_1 \land C_2 \ldots C_m$.
Query: Is $\phi$ nae-satisfiable?

Reduction
1. Construct a new formula $\phi'$ by adding a new variable $s$ to every single clause.
NAESAT

Definition

An assignment to a boolean formula is nae-satisfying, if
1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition

Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,
$$\phi = C_1 \land C_2 \ldots C_m.$$  
Query: Is $\phi$ nae-satisfiable?

Reduction

1. Construct a new formula $\phi'$ by adding a new variable $s$ to every single clause.
2. If $\phi$ is satisfiable, then $\phi'$ is nae-satisfiable.
NAESAT

Definition

An assignment to a boolean formula is nae-satisfying, if
1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition

Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,
$\phi = C_1 \land C_2 \ldots C_m$.

Query: Is $\phi$ nae-satisfiable?

Reduction

1. Construct a new formula $\phi'$ by adding a new variable $s$ to every single clause.
2. If $\phi$ is satisfiable, then $\phi'$ is nae-satisfiable.
3. If $\phi'$ is nae-satisfiable, then $\phi$ must be satisfiable. (Why?)
NAESAT

Definition

An assignment to a boolean formula is nae-satisfying, if

1. It satisfies at least one literal in each clause.
2. It falsifies at least one literal in each clause.

Definition

**Input:** A boolean formula \( \phi \) in CNF form over \( n \) variables and \( m \) clauses, i.e.,
\[
\phi = C_1 \land C_2 \ldots C_m.
\]

**Query:** Is \( \phi \) nae-satisfiable?

Reduction

1. Construct a new formula \( \phi' \) by adding a new variable \( s \) to every single clause.
2. If \( \phi \) is satisfiable, then \( \phi' \) is nae-satisfiable.
3. If \( \phi' \) is nae-satisfiable, then \( \phi \) **must** be satisfiable. (Why?)
4. Thus, SAT \( \leq \) NAESAT.
Using the technique, we can show that NAE4SAT is NP-complete.

To show that NAE3SAT is NP-complete, we simply reduce NAE4SAT to it!

Consider a 4CNF clause $l = (x, y, z, w)$. Argue that $l$ is nae-satisfiable if and only if the following pair of clauses are:

$\left(x, y, s\right)$  
$\left(z, w, \bar{s}\right)$

It follows that NAE3SAT is NP-complete, since 3SAT $\leq$ NAE4SAT $\leq$ NAE3SAT.
NP and NP-completeness
Boolean Circuits
The first NP-complete problem
Satisfiability Problems

NAE3SAT

Reduction

Using the technique, we can show that NAE4SAT is NP-complete. Why?

To show that NAE3SAT is NP-complete, we simply reduce NAE4SAT to it!

Consider a 4CNF clause \( l = (x, y, z, w) \). Argue that \( l \) is nae-satisfiable if and only if the following pair of clauses are:

\[
(x, y, s) \\
(z, w, \bar{s})
\]

It follows that NAE3SAT is NP-complete, since 3SAT \( \leq \) NAE4SAT \( \leq \) NAE3SAT.
Using the technique, we can show that NAE4SAT is NP-complete. Why?

Consider a 4CNF clause $l = (x, y, z, w)$. Argue that $l$ is NAE-satisfiable if and only if the following pair of clauses are:

$$(x, y, s)$$
$$(z, w, \bar{s})$$

It follows that NAE3SAT is NP-complete, since 3SAT $\leq_{P} NAE4SAT \leq_{P} NAE3SAT$. 

1. Using the technique, we can show that NAE4SAT is NP-complete. Why?
Using the technique, we can show that NAE4SAT is **NP-complete**. Why?

To show that NAE3SAT is **NP-complete**, we simply reduce NAE4SAT to it!
Using the technique, we can show that NAE4SAT is **NP-complete**. Why?

To show that NAE3SAT is **NP-complete**, we simply reduce NAE4SAT to it!

Consider a 4CNF clause $l = (x, y, z, w)$. Argue that $l$ is nae-satisfiable if and only if the following pair of clauses are:
NAE3SAT

Reduction

1. Using the technique, we can show that NAE4SAT is **NP-complete**. Why?
2. To show that NAE3SAT is **NP-complete**, we simply reduce NAE4SAT to it!
3. Consider a 4CNF clause $l = (x, y, z, w)$. Argue that $l$ is nae-satisfiable if and only if the following pair of clauses are:

   $$(x, y, s)$$

   $$(z, w, \bar{s})$$

It follows that NAE3SAT is **NP-complete**, since $3SAT \leq NAE4SAT \leq NAE3SAT$. 

NP-completeness
Computational Complexity
NAE3SAT

**Reduction**

1. Using the technique, we can show that NAE4SAT is **NP-complete**. Why?
2. To show that NAE3SAT is **NP-complete**, we simply reduce NAE4SAT to it!
3. Consider a 4CNF clause \( l = (x, y, z, w) \). Argue that \( l \) is nae-satisfiable if and only if the following pair of clauses are:
   \[(x, y, s)\]
   \[(z, w, \bar{s})\]
4. It follows that NAE3SAT is **NP-complete**, since 3SAT \( \leq \) NAE4SAT \( \leq \) NAE3SAT.
MaxSAT

### Definition

**Input:**
A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$ and a number $K \leq m$.

**Query:**
Is there a subset of $K$ or more clauses of $\phi$ which is satisfiable?

### Observations

1. MaxSAT is trivially NP-complete. (Why?)
2. In general, if $k$SAT is NP-complete, so is Max$k$SAT.
3. How about Max$2$SAT?
4. We will show that $\text{NAE } 3$SAT $\leq_{\text{poly}}$ Max$2$SAT.
MaxSAT

Definition

Input:
A boolean formula \( \phi \) in CNF form over \( n \) variables and \( m \) clauses, i.e., \( \phi = C_1 \land C_2 \ldots C_m \) and a number \( K \leq m \).

Query:
Is there a subset of \( K \) or more clauses of \( \phi \) which is satisfiable?

Observations
1. MaxSAT is trivially NP-complete. (Why?)
2. In general, if kSAT is NP-complete, so is MaxkSAT.
3. How about Max2SAT?
4. We will show that NAE3SAT \( \leq \) Max2SAT.
Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses,
MaxSAT

Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$ and a number $K \leq m$. 
MaxSAT

**Definition**

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$ and a number $K \leq m$.

**Query:** Is there a subset of $K$ or more clauses of $\phi$ which is satisfiable?
MaxSAT

**Definition**

*Input:* A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e.,
$\phi = C_1 \land C_2 \ldots C_m$ and a number $K \leq m$.

*Query:* Is there a subset of $K$ or more clauses of $\phi$ which is satisfiable?

**Observations**

1. MaxSAT is trivially NP-complete. (Why?)
2. In general, if $k$SAT is NP-complete, so is Max$k$SAT.
3. How about Max$2$SAT?
4. We will show that NAE$3$SAT $\leq$ Max$2$SAT.
MaxSAT

Definition

**Input:** A boolean formula \( \phi \) in CNF form over \( n \) variables and \( m \) clauses, i.e., \( \phi = C_1 \land C_2 \ldots C_m \) and a number \( K \leq m \).

**Query:** Is there a subset of \( K \) or more clauses of \( \phi \) which is satisfiable?

Observations

1. **MaxSAT is trivially NP-complete.** (Why?)
MaxSAT

Definition

Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$ and a number $K \leq m$.

Query: Is there a subset of $K$ or more clauses of $\phi$ which is satisfiable?

Observations

1. MaxSAT is trivially NP-complete. (Why?)
2. In general, if $k$SAT is NP-complete, so is Max$k$SAT.
MaxSAT

Definition

Input: A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$ and a number $K \leq m$.

Query: Is there a subset of $K$ or more clauses of $\phi$ which is satisfiable?

Observations

1. MaxSAT is trivially NP-complete. (Why?)
2. In general, if $k$SAT is NP-complete, so is Max$k$SAT.
3. How about Max2SAT?
MaxSAT

Definition

**Input:** A boolean formula $\phi$ in CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$ and a number $K \leq m$.

**Query:** Is there a subset of $K$ or more clauses of $\phi$ which is satisfiable?

Observations

1. MaxSAT is trivially \textbf{NP-complete}. (Why?)
2. In general, if $k$SAT is \textbf{NP-complete}, so is Max$k$SAT.
3. How about Max2SAT?
4. We will show that NAE3SAT $\leq$ Max2SAT.
Max2SAT

**Definition**

Input: A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 2 literals and a number $K \leq m$.

Query: Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?

**Reduction**

1. Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses.

2. Consider the clause $l = (x, y, z)$ of the NAE3SAT instance. Replace it with the following set:

   - $(x, y)$
   - $(y, z)$
   - $(x, z)$
   - $(\overline{x}, \overline{y})$
   - $(\overline{y}, \overline{z})$
   - $(\overline{x}, \overline{z})$

3. Set $K = 5 \cdot m$.

4. In argument, note that any assignment satisfies 3 or 5 of the clause set, depending on whether or not it nae-satisfies $l$. 

**NP-completeness**

Computational Complexity
Max2SAT

**Definition**

Input: A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 2 literals and a number $K \leq m$.

Query: Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?

Reduction

1. Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses.
2. Consider the clause $l = (x, y, z)$ of the NAE3SAT instance. Replace it with the following set:
   - $(x, y)$
   - $(y, z)$
   - $(x, z)$
   - $(\overline{x}, \overline{y})$
   - $(\overline{y}, \overline{z})$
   - $(\overline{x}, \overline{z})$
3. Set $K = 5 \cdot m$.
4. In argument, note that any assignment satisfies 3 or 5 of the clause set, depending on whether or not it nae-satisfies $l$. 

Computational Complexity
**Max2SAT**

**Definition**

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses,
Max2SAT

Definition

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 2 literals and a number $K \leq m$. 

**Reduction**

Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses.

Consider the clause $l = (x, y, z)$ of the NAE3SAT instance. Replace it with the following set:

- $(x, y)$
- $(y, z)$
- $(x, z)$
- $(\overline{x}, \overline{y})$
- $(\overline{y}, \overline{z})$
- $(\overline{x}, \overline{z})$

Set $K = 5 \cdot m$.

In argument, note that any assignment satisfies 3 or 5 of the clause set, depending on whether or not it nae-satisfies $l$. 

NP and NP-completeness
Boolean Circuits
The first NP-complete problem
Satisfiability Problems
Max2SAT

**Definition**

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e.,

$$\phi = C_1 \land C_2 \ldots C_m,$$

with each clause having *exactly* 2 literals and a number $K \leq m$.

**Query:** Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?
**Max2SAT**

**Definition**

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 2 literals and a number $K \leq m$.

**Query:** Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?

**Reduction**

Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses. Consider the clause $l = (x, y, z)$ of the NAE3SAT instance. Replace it with the following set:

- $(x, y)$
- $(y, z)$
- $(x, z)$
- $(\bar{x}, \bar{y})$
- $(\bar{y}, \bar{z})$
- $(\bar{x}, \bar{z})$

Set $K = 5 \cdot m$.

In argument, note that any assignment satisfies 3 or 5 of the clause set, depending on whether or not it nae-satisfies $l$. 

---

**NP and NP-completeness**

**Boolean Circuits**

The first NP-complete problem

**Satisfiability Problems**
Max2SAT

**Definition**

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 2 literals and a number $K \leq m$.

**Query:** Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?

**Reduction**

1. Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses.
Max2SAT

Definition

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having exactly 2 literals and a number $K \leq m$.

**Query:** Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?

Reduction

1. Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses.
2. Consider the clause $l = (x, y, z)$ of the NAE3SAT instance. Replace it with the following set:
Max2SAT

Definition

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 2 literals and a number $K \leq m$.

**Query:** Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?

Reduction

1. Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses.
2. Consider the clause $l = (x, y, z)$ of the NAE3SAT instance. Replace it with the following set:

   $$(x, y) \quad (y, z) \quad (x, z)$$
   $$(\bar{x}, \bar{y}) \quad (\bar{y}, \bar{z}) \quad (\bar{x}, \bar{z})$$
Max2SAT

**Definition**

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 2 literals and a number $K \leq m$.

**Query:** Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?

**Reduction**

1. Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses.

2. Consider the clause $l = (x, y, z)$ of the NAE3SAT instance. Replace it with the following set:

   $$(x, y) \quad (y, z) \quad (x, z)$$
   $$(\bar{x}, \bar{y}) \quad (\bar{y}, \bar{z}) \quad (\bar{x}, \bar{z})$$

3. Set $K = 5 \cdot m$. 
Max2SAT

**Definition**

**Input:** A boolean formula $\phi$ in 2CNF form over $n$ variables and $m$ clauses, i.e., $\phi = C_1 \land C_2 \ldots C_m$, with each clause having *exactly* 2 literals and a number $K \leq m$.

**Query:** Is there a subset of $\phi$ with cardinality at least $K$, which is satisfiable?

**Reduction**

1. Assume that you are given an instance of NAE3SAT over $n$ variables and $m$ clauses.
2. Consider the clause $l = (x, y, z)$ of the NAE3SAT instance. Replace it with the following set:
   
   $$(x, y) \quad (y, z) \quad (x, z)$$
   $$(\bar{x}, \bar{y}) \quad (\bar{y}, \bar{z}) \quad (\bar{x}, \bar{z})$$

3. Set $K = 5 \cdot m$.

4. In argument, note that any assignment satisfies 3 or 5 of the clause set, depending on whether or not it nae-satisfies $l$. 

NP-completeness

**Computational Complexity**
Integer Programming (IP)

Definition

Input:
An integer matrix $A$ $m \times n$ and an integer vector $b$ $m \times 1$.

Query:
Is there a lattice point $r \in \mathbb{Z}^n$, such that $A \cdot r \geq b$?

Observation

It is non-trivial to show that IP is in $NP$. Hence, we will focus on a restriction called $0/1$ IP, where each component of the vector $r$ is required to be 0 or 1.
Integer Programming (IP)

**Definition**

Input: An integer matrix $A_{m \times n}$ and an integer vector $b_{m \times 1}$.

Query: Is there a lattice point $r \in \mathbb{Z}^n$, such that $A \cdot r \geq b$?

Observation: It is non-trivial to show that IP is in $\text{NP}$. Hence, we will focus on a restriction called $0/1\text{IP}$, where each component of the vector $r$ is required to be 0 or 1.
Definition

**Input:** An integer matrix $A_{m \times n}$ and an integer vector $b_{m \times 1}$. 
**Definition**

**Input:** An integer matrix $A_{m \times n}$ and an integer vector $b_{m \times 1}$.

**Query:** Is there a lattice point $r \in \mathbb{Z}^n_+$, such that $A \cdot r \geq b$?
Definition

Input: An integer matrix $A_{m \times n}$ and an integer vector $b_{m \times 1}$.

Query: Is there a lattice point $r \in \mathbb{Z}^n_+$, such that $A \cdot r \geq b$?

Observation

It is non-trivial to show that IP is in $NP$. Hence, we will focus on a restriction called $0/1$ IP, where each component of the vector $r$ is required to be 0 or 1.
Integer Programming (IP)

**Definition**

**Input:** An integer matrix $A_{m \times n}$ and an integer vector $b_{m \times 1}$.

**Query:** Is there a lattice point $r \in \mathbb{Z}^n_+$, such that $A \cdot r \geq b$?

**Observation**

*It is non-trivial to show that IP is in NP.*
Integer Programming (IP)

**Definition**

**Input:** An integer matrix $A_{m \times n}$ and an integer vector $b_{m \times 1}$.

**Query:** Is there a lattice point $r \in \mathbb{Z}_n^+$, such that $A \cdot r \geq b$?

**Observation**

*It is non-trivial to show that IP is in NP.*

*Hence, we will focus on a restriction called 0/1 IP, where each component of the vector $r$ is required to be 0 or 1.*
Theorem. $0/1$ IP is NP-complete.

Proof. $0/1$ IP is clearly in NP.

We will show that 3SAT $\leq_{0/1}$ IP.

Take the clause $l = (x, \overline{y}, z)$.

Replace it with the constraint: $c: x + (1 - y) + z \geq 1$.

Argue that if $l$ has a satisfying assignment then so does $c$ and vice versa.

The theorem follows.

Observations: Integer Programming rivals 3SAT in terms of versatility.
Theorem

0/1 IP

Theorem

0/1 IP is NP-complete.

Proof
1. IP is clearly in NP.
2. We will show that 3SAT ≤ IP.
3. Take the clause \( l: (x, \overline{y}, z) \).
4. Replace it with the constraint: \( c: x + (1 - y) + z \geq 1 \).
5. Argue that if \( l \) has a satisfying assignment then so does \( c \) and vice versa.
6. The theorem follows.

Observations
Integer Programming rivals 3SAT in terms of versatility.
Theorem

0/1 IP is **NP-complete.**
Theorem

0/1 IP is NP-complete.

Proof

Argue that if \( l \) has a satisfying assignment then \( c \) and vice versa.

Theorem follows.
Theorem

0/1 IP is NP-complete.

Proof

1. 0/1 IP is clearly in NP.
0/1 IP is NP-complete.

Proof

1. 0/1 IP is clearly in NP.
2. We will show that 3SAT ≤ 0/1 IP.
Theorem

0/1 IP is NP-complete.

Proof

1. 0/1 IP is clearly in NP.
2. We will show that 3SAT \leq 0/1 IP.
3. Take the clause \( l : (x, \overline{y}, z) \).
Theorem

0/1 IP is **NP-complete**.

Proof

1. 0/1 IP is clearly in **NP**.
2. We will show that 3SAT ≤ 0/1 IP.
3. Take the clause \( l : (x, \bar{y}, z) \).
4. Replace it with the constraint: \( c : x + (1 - y) + z \geq 1 \).
Theorem

0/1 IP is NP-complete.

Proof

1. 0/1 IP is clearly in NP.
2. We will show that 3SAT \( \leq \) 0/1 IP.
3. Take the clause \( l : (x, \bar{y}, z) \).
4. Replace it with the constraint: \( c : x + (1 - y) + z \geq 1 \).
5. Argue that if \( l \) has a satisfying assignment then do does \( c \).
Theorem

0/1 IP is NP-complete.

Proof

1. 0/1 IP is clearly in NP.
2. We will show that 3SAT ≤ 0/1 IP.
3. Take the clause \( l : (x, \overline{y}, z) \).
4. Replace it with the constraint: \( c : x + (1 - y) + z \geq 1 \).
5. Argue that if \( l \) has a satisfying assignment then do does \( c \) and vice versa.
Theorem

0/1 IP is NP-complete.

Proof

1. 0/1 IP is clearly in NP.
2. We will show that 3SAT ≤ 0/1 IP.
3. Take the clause \( l : (x, \bar{y}, z) \).
4. Replace it with the constraint: \( c : x + (1 - y) + z \geq 1 \).
5. Argue that if \( l \) has a satisfying assignment then do does \( c \) and vice versa.
6. The theorem follows.
Theorem

0/1 IP is **NP-complete**.

Proof

1. 0/1 IP is clearly in **NP**.
2. We will show that 3SAT \(\leq\) 0/1 IP.
3. Take the clause \(l: (x, \overline{y}, z)\).
4. Replace it with the constraint: \(c: x + (1 - y) + z \geq 1\).
5. Argue that if \(l\) has a satisfying assignment then so does \(c\) and vice versa.
6. The theorem follows.

Observations

*Integer Programming rivals 3SAT in terms of versatility.*