


Aleks Ignjatovi

School of Computer Science and En
University of New South Wales

11. INTRACTABILITY

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.
- This means that there exists a natural number $k$ (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where $n$ is the size of the input.

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.
- This means that there exists a natural number $k$ (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where $n$ is the size of the input.
- A $d$

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.
- This means that there exists a natural number $k$ (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where $n$ is the size of the input.
- A $d$
- Exa
  - "*Input number n is a prime number.*

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.
- This means that there exists a natural number $k$ (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where $n$ is the size of the input.
- A $d$
- Exa
  - "*Input number n is a prime number.*
  - "*Input graph G is connected.*"

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.
- This means that there exists a natural number $k$ (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where $n$ is the size of the input.
- A $d$
- Exa
  - "*Input number n is a prime number.*
  - "*Input graph G is connected.*"
  - "*Input graph G has a cycle containing all ver*

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.
- This means that there exists a natural number $k$ (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where $n$ is the size of the input.
- A $d$
- Exa
  - "*Input number n is a prime number.*
  - "*Input graph G is connected.*"
  - "*Input graph G has a cycle containing all ver*
- We say that a decision problem $A$ is in *polynomial time* if there exists a polynomial time algorithm which solves it.

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.
- This means that there exists a natural number $k$ (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where $n$ is the size of the input.
- A $d$
- Exa
  - "*Input number n is a prime number.*
  - "*Input graph G is connected.*"
  - "*Input graph G has a cycle containing all ver*
- We say that a decision problem $A$ is in *polynomial time* if there exists a polynomial time algorithm which solves it.
- Thus, given an input $x$, such an algorithm outputs $YES$ for all $x$ which satisfy $A$ and outputs $NO$ for all $x$ which do not satisfy $A$.

# Feasibility of Algorithms

- We say that a (sequential) algorithm is *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.
- This means that there exists a natural number $k$ (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where $n$ is the size of the input.
- A $d$
- Exa
  - "*Input number n is a prime number.*
  - "*Input graph G is connected.*"
  - "*Input graph G has a cycle containing all ver*
- We say that a decision problem $A$ is in *polynomial time* if there exists a polynomial time algorithm which solves it.
- Thus, given an input $x$, such an algorithm outputs $YES$ for all $x$ which satisfy $A$ and outputs $NO$ for all $x$ which do not satisfy $A$.
- We denote this by $A \in \mathbf{P}$.

# Feasibility of Algorithms

- What is the *length* of an input?

# Feasibility of Algorithms



- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.

# Feasibility of Algorithms



- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.
- Exa

# Feasibility of Algorithms



- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.
- Exa

# Feasibility of Algorithms



- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.
- Exa
  - https://eduassistpro.github.i
  - As we will see, definition of polynomial time computability is quite robust with respect to how we represent inputs.

# Feasibility of Algorithms



- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.
- Exa
  - https://eduassistpro.github.i
  - As we will see, definition of polynomial time computability is quite robust with respect to how we represent inputs.
  - For example, we could also define the length of an i number of digits in the decimal representatio

# Feasibility of Algorithms



- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.
- Exa

  - As we will see, definition of polynomial time computability is quite robust with respect to how we represent inputs.
  - For example, we could also define the length of an i number of digits in the decimal representatio

  - This can only change the constants involved in t $T(n) = O(n^k)$ but not the asymptotic bound.

# Feasibility of Algorithms

- If input is a weighted graph $G$, then $G$ can be described by giving for each vertex $v_i$ a list of edges incident to $v_i$ together with their (integer) weights, represented in binary.

# Feasibility of Algorithms

- If input is a weighted graph $G$, then $G$ can be described by giving for each vertex $v_i$ a list of edges incident to $v_i$ together with their (integer) weights, represented in binary.

- Alte

# Feasibility of Algorithms

- If input is a weighted graph $G$, then $G$ can be described by giving for each vertex $v_i$ a list of edges incident to $v_i$ together with their (integer) weights, represented in binary.
- Alte
- If the i
the representation of the graph.

# Feasibility of Algorithms

- If input is a weighted graph $G$, then $G$ can be described by giving for each vertex $v_i$ a list of edges incident to $v_i$ together with their (integer) weights, represented in binary.
- Alte
- If the i
the representation of the graph.
- However, since we are interested only in whether the al
polynomial time and not in the particular degree of the p
such a run time, this does not matter.

# Feasibility of Algorithms

- If input is a weighted graph $G$, then $G$ can be described by giving for each vertex $v_i$ a list of edges incident to $v_i$ together with their (integer) weights, represented in binary.
- Alte
- If the i
the representation of the graph.
- However, since we are interested only in whether the al
polynomial time and not in the particular degree of the p
such a run time, this does not matter.
- In fact, every precise description without artificial redundancies will do.

# Feasibility of Algorithms

- We say that a decision problem $A(x)$ is in *non-deterministic polynomial time*, denoted by $A \in \mathbf{NP}$, if:

# Feasibility of Algorithms

- We say that a decision problem $A(x)$ is in *non-deterministic polynomial time*, denoted by $A \in \mathbf{NP}$, if:
  1. there exists a problem $B(x, y)$ such that for every input $x$, $A(x)$ is true just in case there exists $y$ such that $B(x, y)$ is true

  and

# Feasibility of Algorithms

- We say that a decision problem $A(x)$ is in *non-deterministic polynomial time*, denoted by $A \in \mathbf{NP}$, if:
  1. there exists a problem $B(x, y)$ such that for every input $x$, $A(x)$ is true just in case there exists $y$ such that $B(x, y)$ is true and
  2. $B$ can be solved in polynomial time.

# Feasibility of Algorithms

- We say that a decision problem $A(x)$ is in *non-deterministic polynomial time*, denoted by $A \in \mathbf{NP}$, if:
  1. there exists a problem $B(x, y)$ such that for every input $x$, $A(x)$ is true just in case there exists $y$ such that $B(x, y)$ is true and
  2. $B$ can be solved in polynomial time.
- We call $y$ a *certificate* for $x$.

# Feasibility of Algorithms

- We say that a decision problem $A(x)$ is in *non-deterministic polynomial time*, denoted by $A \in \mathbf{NP}$, if:
  1. there exists a problem $B(x, y)$ such that for every input $x$, $A(x)$ is true just in case there exists $y$ such that $B(x, y)$ is true and
  2. $B$ can be computed in polynomial time (in the length of $x$).
- We call $y$ a *certificate* for $x$.
- For example, decision problem "integer $x$ is composite" is in $\mathbf{NP}$ because $A(x)$ is true just in case there exists integer $y$ such that $1 < y < x$ and $x$ is divisible by $y$" is true.

# Feasibility of Algorithms

- We say that a decision problem $A(x)$ is in *non-deterministic polynomial time*, denoted by $A \in \mathbf{NP}$, if:
  1. there exists a problem $B(x, y)$ such that for every input $x$, $A(x)$ is true just in case there exists $y$ such that $B(x, y)$ is true and
  2. $y$ is at most $n^k$ for some $k$.
- We call $y$ a *certificate* for $x$.
- For example, decision problem "integer $x$ is composite" is in $\mathbf{NP}$ because $A(x)$ is true just in case there exists integer $y$ such that $1 < y < x$ and $x$ is divisible by $y$" is true.
- Clearly, the problem "$x$ is divisible by $y$" is decidable by an algorithm which runs in time polynomial in the length of $x$ only.

# Feasibility of Algorithms

- We say that a decision problem $A(x)$ is in *non-deterministic polynomial time*, denoted by $A \in \mathbf{NP}$, if:
  1. there exists a problem $B(x, y)$ such that for every input $x$, $A(x)$ is true just in case there exists $y$ such that $B(x, y)$ is true and
  2. $B(x, y)$ can be decided in polynomial time in the length of $x$.
- We call $y$ a *certificate* for $x$.
- For example, decision problem "integer $x$ is composite" is in $\mathbf{NP}$ because $A(x)$ is true just in case there exists integer $y$ such that $1 < y < x$ and $x$ is divisible by $y$" is true.
- Clearly, the problem "$x$ is divisible by $y$" is decidable by an algorithm which runs in time polynomial in the length of $x$ only.
- In fact, "integer $x$ is not prime" is actually decidable in (deterministic) polynomial time, but this is a hard theorem to prove.

# Feasibility of Algorithms

Examples of $NP$-decision problems:

# Feasibility of Algorithms

Examples of $NP$-decision problems:

- (Vertex Cover) Instance: a graph $G$ and an integer $k$. Problem: "There exists a subset $U$ consisting of at most $k$ vertices of $G$ (called a vertex cover of $G$) such that each edge has at least one end belonging to $U$."

# Feasibility of Algorithms

Examples of $NP$-decision problems:

- (Vertex Cover) Instance: a graph $G$ and an integer $k$. Problem: "There exists a subset $U$ consisting of at most $k$ vertices of $G$ (called a vertex cover of $G$) such that each edge has at least one end belonging to $U$."

  Clearly, given a subset of vertices $U$ we can determine in polynomial time if $U$ is a ver

# Feasibility of Algorithms

Examples of $NP$-decision problems:

- (Vertex Cover) Instance: a graph $G$ and an integer $k$. Problem: "There exists a subset $U$ consisting of at most $k$ vertices of $G$ (called a vertex cover of $G$) such that each edge has at least one end belonging to $U$."
  Clearly, given a subset of vertices $U$ we can determine in polynomial time if $U$ is a ver

- (SA
  whe
  negations, for example

  $$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg \qquad _5)$$

# Feasibility of Algorithms

Examples of $NP$-decision problems:

- (Vertex Cover) Instance: a graph $G$ and an integer $k$. Problem: "There exists a subset $U$ consisting of at most $k$ vertices of $G$ (called a vertex cover of $G$) such that each edge has at least one end belonging to $U$."

  Clearly, given a subset of vertices $U$ we can determine in polynomial time if $U$ is a ver

- (SA whe

  negations, for example

  $$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg \qquad 5)$$

  Problem: "There exists an evaluation of the propositi makes the formula true".

# Feasibility of Algorithms

Examples of $NP$-decision problems:

- (Vertex Cover) Instance: a graph $G$ and an integer $k$. Problem: "There exists a subset $U$ consisting of at most $k$ vertices of $G$ (called a vertex cover of $G$) such that each edge has at least one end belonging to $U$."

  Clearly, given a subset of vertices $U$ we can determine in polynomial time if $U$ is a ver

- (SA whe

  negations, for example

  $$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg \qquad 5)$$

  Problem: "There exists an evaluation of the propositi makes the formula true".

  Clearly, given an evaluation of the propositional variables one can determine in polynomial time if the formula is true for such an evaluation.

# Feasibility of Algorithms

Examples of $NP$-decision problems:

- (Vertex Cover) Instance: a graph $G$ and an integer $k$. Problem: "There exists a subset $U$ consisting of at most $k$ vertices of $G$ (called a vertex cover of $G$) such that each edge has at least one end belonging to $U$."

  Clearly, given a subset of vertices $U$ we can determine in polynomial time if $U$ is a ver

- (SA whe negations, for example $C_2 \wedge \cdots \wedge C_r$

  $$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg \quad _5)$$

  Problem: "There exists an evaluation of the propositi makes the formula true".

  Clearly, given an evaluation of the propositional variables one can determine in polynomial time if the formula is true for such an evaluation.

- If each clause $C_i$ involves exactly three variables we call such decision problem 3SAT.

# Polynomial Reductions

- As we have mentioned, for example, the decision problem "integer $n$ is not prime" is obviously in NP, but it has been proved in 2002 that it is also in P.

# Polynomial Reductions

- As we have mentioned, for example, the decision problem "integer $n$ is not prime" is obviously in NP, but it has been proved in 2002 that it is also in P. (This is a famous and unexpected result, proved by Indian computer scientists Agrawal, Kayal and Saxena.)

# Polynomial Reductions

- As we have mentioned, for example, the decision problem "integer $n$ is not prime" is obviously in NP, but it has been proved in 2002 that it is also in P. (This is a famous and unexpected result, proved by Indian computer scientists Agrawal, Kayal and Saxena.)
- Is it the case that *every* NP problem is also in P??

# Polynomial Reductions

- As we have mentioned, for example, the decision problem "integer $n$ is not prime" is obviously in NP, but it has been proved in 2002 that it is also in P. (This is a famous and unexpected result, proved by Indian computer scientists Agrawal, Kayal and Saxena.)



- Is it the case that *every* NP problem is also in P??

- For e... aluations of $n$ pr... is not
these propositional variables makes $\Phi$ true?

# Polynomial Reductions

- As we have mentioned, for example, the decision problem "integer $n$ is not prime" is obviously in NP, but it has been proved in 2002 that it is also in P. (This is a famous and unexpected result, proved by Indian computer scientists Agrawal, Kayal and Saxena.)
- Is it the case that *every* NP problem is also in P??
- For e aluations of $n$ pr is not these propositional variables makes $\Phi$ true?
- Intuitively, this should not be the case; determining if s should be a harder problem than simply checking if a giv such a formula true.

# Polynomial Reductions

- As we have mentioned, for example, the decision problem "integer $n$ is not prime" is obviously in NP, but it has been proved in 2002 that it is also in P. (This is a famous and unexpected result, proved by Indian computer scientists Agrawal, Kayal and Saxena.)
- Is it the case that *every* NP problem is also in P??
- For e... ...aluations of $n$ pr... is not these propositional variables makes $\Phi$ true?
- Intuitively, this should not be the case; determining if s... should be a harder problem than simply checking if a giv... such a formula true.
- However, so far, no one has been able to prove (or disprove) that this is indeed the case, despite a huge effort of very many very famous people!!

# Polynomial Reductions

- As we have mentioned, for example, the decision problem "integer $n$ is not prime" is obviously in NP, but it has been proved in 2002 that it is also in P. (This is a famous and unexpected result, proved by Indian computer scientists Agrawal, Kayal and Saxena.)
- Is it the case that *every* NP problem is also in P??
- For e... ...aluations of $n$ pr... is not these propositional variables makes $\Phi$ true?
- Intuitively, this should not be the case; determining if s... should be a harder problem than simply checking if a giv... such a formula true.
- However, so far, no one has been able to prove (or disprove) that this is indeed the case, despite a huge effort of very many very famous people!!
- Conjecture that $NP$ is a strictly larger class of decision problems than $P$ is known as "$P \neq NP$" hypothesis, and it is widely believed that it is one of the hardest open problems in the whole of Mathematics!!

# Polynomial Reductions

- Let $U$ and $V$ be two decision problems. We say that $U$ is polynomially reducible to $V$ if and only if there exists a function $f(x)$ such that:

# Polynomial Reductions

- Let $U$ and $V$ be two decision problems. We say that $U$ is polynomially reducible to $V$ if and only if there exists a function $f(x)$ such that:
  1. $f(x)$ maps instances of $U$ into instances of $V$;

# Polynomial Reductions

- Let $U$ and $V$ be two decision problems. We say that $U$ is polynomially reducible to $V$ if and only if there exists a function $f(x)$ such that:
  1. $f(x)$ maps instances of $U$ into instances of $V$;
  2. For every instance $x$ of $U$ we have that $U(x)$ is true just in case $f(x)$ is an instance of $V$ such that $V(f(x))$ is true.

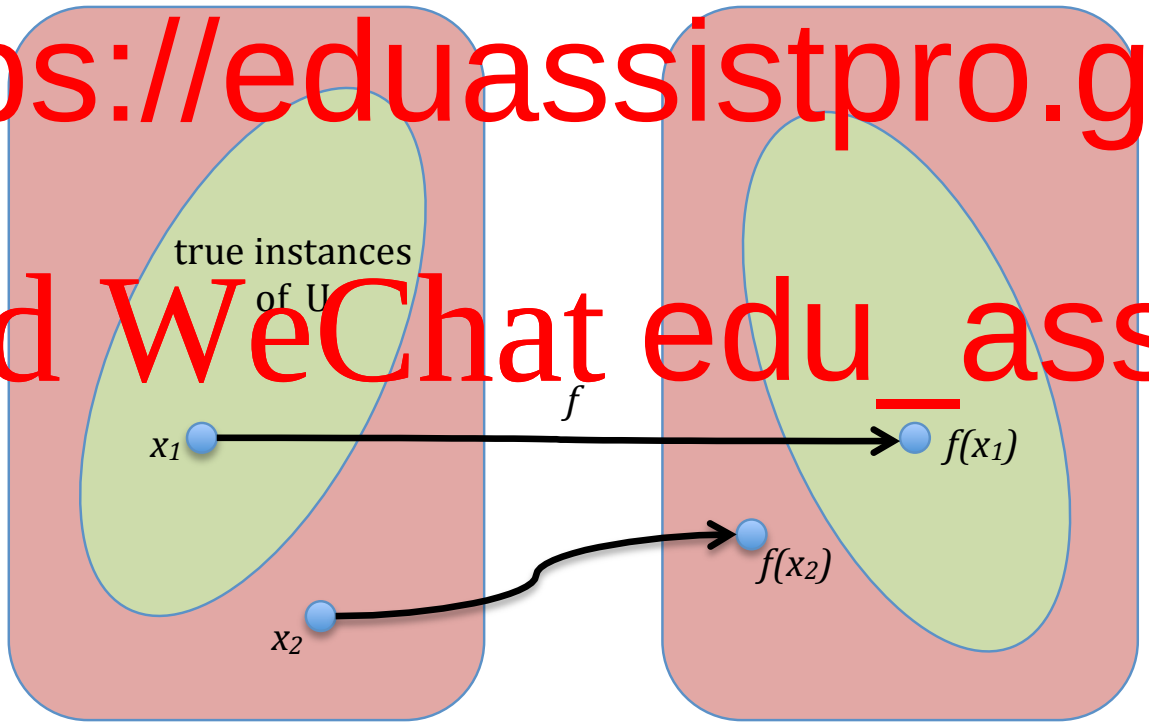
# Polynomial Reductions

- Let $U$ and $V$ be two decision problems. We say that $U$ is polynomially reducible to $V$ if and only if there exists a function $f(x)$ such that:
  1. $f(x)$ maps instances of $U$ into instances of $V$;
  2. For every instance $x$ of $U$ we have that $U(x)$ is true just in case $f(x)$ is an instance of $V$ such that $V(f(x))$ is true.
  3. $f(x)$ is computable by a polynomial time algorithm.

# Polynomial Reductions

- Let $U$ and $V$ be two decision problems. We say that $U$ is polynomially reducible to $V$ if and only if there exists a function $f(x)$ such that:
  1. $f(x)$ maps instances of $U$ into instances of $V$;
  2. For every instance $x$ of $U$ we have that $U(x)$ is true just in case $f(x)$ is an instance of $V$ such that $V(f(x))$ is true.
  3. $f(x)$ is computable by a polynomial time algorithm.





# Polynomial Reductions

Example of a polynomial reduction:

# Polynomial Reductions

Example of a polynomial reduction:

- Every instance of SAT is polynomially reducible to an instance of 3SAT.

# Polynomial Reductions

Example of a polynomial reduction:

- Every instance of SAT is polynomially reducible to an instance of 3SAT.
- We introduce more propositional variables and replace every clause by a conjunction of several clauses.

# Polynomial Reductions

Example of a polynomial reduction:

- Every instance of SAT is polynomially reducible to an instance of 3SAT.
- We introduce more propositional variables and replace every clause by a conjunction of several clauses.
- For example, we replace clause

$$\_ \quad \_ \quad \_ \tag{1}$$

# Polynomial Reductions

Example of a polynomial reduction:

- Every instance of SAT is polynomially reducible to an instance of 3SAT.
- We introduce more propositional variables and replace every clause by a conjunction of several clauses.
- For example, we replace clause

$$\underbrace{P_1 \vee \neg P_2 \vee Q_1}_{(1)} \tag{1}$$

with the variables $Q_1, Q_2, Q_3$:

$$\underbrace{(P_1 \vee \neg P_2 \vee Q_1)}_{(1)} \wedge \underbrace{(\neg Q_1 \vee \neg P_3 \vee Q_2)}_{(2)} \wedge \underbrace{(\neg Q \quad P_6)}_{(3)} \tag{2}$$

# Polynomial Reductions

Example of a polynomial reduction:

- Every instance of SAT is polynomially reducible to an instance of 3SAT.
- We introduce more propositional variables and replace every clause by a conjunction of several clauses.
- For example, we replace clause

$$\underbrace{(P_1 \vee \neg P_2 \vee Q_1)}_{(1)} \wedge \underbrace{(\neg Q_1 \vee \neg P_3 \vee Q_2)}_{(2)} \wedge \underbrace{(\neg Q \quad P_6)}_{(t)} \quad (??)$$

with t variables $Q_1, Q_2, Q_3$:

- Easy to verify that if an evaluation of $P_i's$ makes (**??**) true, then such evaluation can be extended by an evaluation of $Q_j's$ such that (**??**) is also true

# Polynomial Reductions

Example of a polynomial reduction:

- Every instance of SAT is polynomially reducible to an instance of 3SAT.
- We introduce more propositional variables and replace every clause by a conjunction of several clauses.
- For example, we replace clause

$$\underbrace{(P_1 \vee \neg P_2 \vee Q_1)}_{(1)} \wedge \underbrace{(\neg Q_1 \vee \neg P_3 \vee Q_2)}_{(2)} \wedge \underbrace{(\neg Q \quad P_6)}_{(3)} \quad (1)$$

with t
variables $Q_1, Q_2, Q_3$:

- Easy to verify that if an evaluation of $P_i's$ makes (**??**) true, then such evaluation can be extended by an evaluation of $Q_j's$ such that (**??**) is also true
- and vice versa: every evaluation which makes (**??**) true also makes (**??**) true.

# Polynomial Reductions

Example of a polynomial reduction:

- Every instance of SAT is polynomially reducible to an instance of 3SAT.
- We introduce more propositional variables and replace every clause by a conjunction of several clauses.
- For example, we replace clause

$$\underbrace{(P_1 \vee \neg P_2 \vee Q_1)}_{(1)} \wedge \underbrace{(\neg Q_1 \vee \neg P_3 \vee Q_2)}_{(2)} \wedge \underbrace{(\neg Q \quad P_6)}_{(1)}$$

with t variables $Q_1, Q_2, Q_3$:

- Easy to verify that if an evaluation of $P_i's$ makes (??) true, then such evaluation can be extended by an evaluation of $Q_j's$ such that (??) is also true
- and vice versa: every evaluation which makes (??) true also makes (??) true.
- Clearly, (??) can be obtained from (??) using a simple polynomial time algorithm.

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:
  1. for every instance $x$ of $U$, $f(x)$ is a propositional formula $\Phi_x$;

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:
  1. for every instance $x$ of $U$, $f(x)$ is a propositional formula $\Phi_x$;
  2.

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:
  1. for every instance $x$ of $U$, $f(x)$ is a propositional formula $\Phi_x$;
  2.
- An N m is poly

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:
  1. for every instance $x$ of $U$, $f(x)$ is a propositional formula $\Phi_x$;
  2.
- An N m is poly
- Thus, Cook's Theorem says that SAT is NP complete.

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:
  1. for every instance $x$ of $U$, $f(x)$ is a propositional formula $\Phi_x$;
  2.
- An N m is poly
- Thus, Cook's Theorem says that SAT is NP complete.
- NP complete problems are in a sense universal: if we had a solves one single NP complete problem $U$, t to solve every other NP problem:

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:
  1. for every instance $x$ of $U$, $f(x)$ is a propositional formula $\Phi_x$;
  2.
- An N m is poly
- Thus, Cook's Theorem says that SAT is NP complete.
- NP complete problems are in a sense universal: if we had a solves one single NP complete problem $U$, t to solve every other NP problem:
- A solution of an instance $x$ of any other NP problem $V$ could simply be obtained by:

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:
  1. for every instance $x$ of $U$, $f(x)$ is a propositional formula $\Phi_x$;
  2.
- An N m is poly
- Thus, Cook's Theorem says that SAT is NP complete.
- NP complete problems are in a sense universal: if we had a solves one single NP complete problem $U$, t to solve every other NP problem:
- A solution of an instance $x$ of any other NP problem $V$ could simply be obtained by:
  1. computing in polynomial time the reduction $f(x)$ of $V$ to $U$,

# Cook's Theorem

- **Theorem:** Every NP problem is polynomially reducible to the SAT problem.
- This means that for every NP decision problem $U(x)$ there exists a polynomial time computable function $f(x)$ such that:
  1. for every instance $x$ of $U$, $f(x)$ is a propositional formula $\Phi_x$;
  2.
- An N m is poly
- Thus, Cook's Theorem says that SAT is NP complete.
- NP complete problems are in a sense universal: if we had a solves one single NP complete problem $U$, t to solve every other NP problem:
- A solution of an instance $x$ of any other NP problem $V$ could simply be obtained by:
  1. computing in polynomial time the reduction $f(x)$ of $V$ to $U$,
  2. then running the algorithm that solves $U$ on instance $f(x)$.

# Polynomial Reductions

- So NP complete problems are the hardest NP problems - a polynomial time algorithm for solving an NP complete problem would make every other NP problem also solvable in polynomial time.

# Polynomial Reductions

- So NP complete problems are the hardest NP problems - a polynomial time algorithm for solving an NP complete problem would make every other NP problem also solvable in polynomial time.
- But it is true what most people believe, i.e., that $P \neq NP$, then there cannot be any polynomial time algorithms for solving an NP complete prob

# Polynomial Reductions

- So NP complete problems are the hardest NP problems - a polynomial time algorithm for solving an NP complete problem would make every other NP problem also solvable in polynomial time.
- But if it is true what most people believe, i.e., that $P \neq NP$, then there cannot be any polynomial time algorithms for solving an NP complete prob
- So wh

# Polynomial Reductions

- So NP complete problems are the hardest NP problems - a polynomial time algorithm for solving an NP complete problem would make every other NP problem also solvable in polynomial time.
- But it it is true what most people believe, i.e., that $P \neq NP$, then there cannot be any polynomial time algorithms for solving an NP complete prob
- So wh
- Maybe SAT is not that important - why should we care a propositional formulas?

# Polynomial Reductions

- So NP complete problems are the hardest NP problems - a polynomial time algorithm for solving an NP complete problem would make every other NP problem also solvable in polynomial time.
- But it it is true what most people believe, i.e., that $P \neq NP$, then there cannot be any polynomial time algorithms for solving an NP complete prob
- So wh
- Maybe SAT is not that important - why should we care a propositional formulas?
- Maybe NP complete problems only have theoretica practical relevance??

# Polynomial Reductions

- So NP complete problems are the hardest NP problems - a polynomial time algorithm for solving an NP complete problem would make every other NP problem also solvable in polynomial time.
- But it is true what most people believe, i.e., that $P \neq NP$, then there cannot be any polynomial time algorithms for solving an NP complete prob
- So wh
- Maybe SAT is not that important - why should we care a propositional formulas?
- Maybe NP complete problems only have theoretica practical relevance??
- Unfortunately, this cannot be farthest from the truth!

# Polynomial Reductions

- So NP complete problems are the hardest NP problems - a polynomial time algorithm for solving an NP complete problem would make every other NP problem also solvable in polynomial time.
- But it it is true what most people believe, i.e., that $P \neq NP$, then there cannot be any polynomial time algorithms for solving an NP complete prob
- So wh
- Maybe SAT is not that important - why should we care a propositional formulas?
- Maybe NP complete problems only have theoretica practical relevance??
- Unfortunately, this cannot be farthest from the truth!
- A vast number of practically important decision problems are NP complete!

# NP complete problems are everywhere!

- **Traveling Salesman Problem**

# NP complete problems are everywhere!

- **Traveling Salesman Problem**
  - Instance:

# NP complete problems are everywhere!

- **Traveling Salesman Problem**
  1. Instance:
     1. a map, i.e., a weighted graph with locations as vertices and with

# NP complete problems are everywhere!

- **Traveling Salesman Problem**
  1. Instance:
     1. a map, i.e., a weighted graph with locations as vertices and with
  2.

# NP complete problems are everywhere!

- **Traveling Salesman Problem**



1. Instance:
   1. a map, i.e., a weighted graph with locations as vertices and with
2. Problem: Is there a tour along the edges visiting e vertex) exactly once and returning to the starti total length of the tour at most $L$?

# NP complete problems are everywhere!

- **Traveling Salesman Problem**
  1. Instance:
     1. a map, i.e., a weighted graph with locations as vertices and with
  2. Problem: Is there a tour along the edges visiting e vertex) exactly once and returning to the starti total length of the tour at most $L$?



- Think of a mailman which has to deliver mail to several addresses and then return to the post office. Can he do it while traveling less than $L$ kilometres in total?

# NP complete problems are everywhere!

- **Register Allocation Problem**

# NP complete problems are everywhere!

- **Register Allocation Problem**
  1. Instance:

# NP complete problems are everywhere!

- **Register Allocation Problem**
  1. Instance:
     -

# NP complete problems are everywhere!

- **Register Allocation Problem**
  1. Instance:
     - 
     - 

same step of the execution of the program;

# NP complete problems are everywhere!

- **Register Allocation Problem**



1. Instance:
   - 
   - 



   same step of the execution of the program;
3. The number of registers $K$ of the

# NP complete problems are everywhere!

- **Register Allocation Problem**
  1. Instance:
     - 
     - 
     - same step of the execution of the program;
     - The number of registers $K$ of the
  2. Problem: is it possible to assign variables to regis
     both vertices assigned to the same register.

# NP complete problems are everywhere!



- **Register Allocation Problem**
  1. Instance:
     - 
     - 
     - same step of the execution of the program;
     - The number of registers $K$ of the
  2. Problem: is it possible to assign variables to regis
     both vertices assigned to the same register.





- In graph theoretic terms: Is it possible to color the vertices of a graph $G$ with at most $K$ colors so that no edge has both vertices of the same color.

# NP complete problems are everywhere!

- Set CoverProblem

# NP complete problems are everywhere!



- **Set CoverProblem**
  - Assume you want to buy DVDs, each with one out of $N$ movie that you

# NP complete problems are everywhere!



- **Set CoverProblem**
  - Assume you want to buy DVDs, each with one out of $N$ movie that you
  - ice $p_i$.

# NP complete problems are everywhere!



- **Set CoverProblem**
  - Assume you want to buy DVDs, each with one out of $N$ movie that you
  - 
  - ice $p_i$.
  - some bundles may contain several movies tha
  - For each bundle $B_i$ you have a list $l_i$

# NP complete problems are everywhere!



- **Set CoverProblem**
  - Assume you want to buy DVDs, each with one out of $N$ movie that you
  - 
  - ice $p_i$.
  - some bundles may contain several movies tha
  - For each bundle $B_i$ you have a list $l_i$
  - Your goal is to choose a set of bundles which togeth
    movies you want to buy and which has the smalle

# NP complete problems are everywhere!



- **Set CoverProblem**
  - Assume you want to buy DVDs, each with one out of $N$ movie that you
  - 
  - ice $p_i$.
  - some bundles may contain several movies tha
  - For each bundle $B_i$ you have a list $l_i$
  - Your goal is to choose a set of bundles which togeth
    movies you want to buy and which has the smalle



- As we will see, many other practically important problems are NP complete.

# NP hard problems

- Let $A$ be a problem and assume that we have a 'black box" device (also called "an oracle") which for every input $x$ instantaneously computes $A(x)$.

# NP hard problems

- Let $A$ be a problem and assume that we have a 'black box" device (also called "an oracle") which for every input $x$ instantaneously computes $A(x)$.
- We consider algorithms which are *polynomial time in* $A$. This means algorithms which run in polynomial time in the length of the input and which, besides the usual computational steps, can also consult a "black box" for solving problem $A(y)$ for any intermediate output $y$.

# NP hard problems

- Let $A$ be a problem and assume that we have a 'black box" device (also called "an oracle") which for every input $x$ instantaneously computes $A(x)$.
- We consider algorithms which are *polynomial time in* $A$. This means algorithms which run in polynomial time in the length of the input and which, besides the usual computational steps, can also consult a "black box" for solving problem $A(y)$ for any intermediate output $y$.
- We s me in $A$, i.e. orithm whic

# NP hard problems

- Let $A$ be a problem and assume that we have a 'black box" device (also called "an oracle") which for every input $x$ instantaneously computes $A(x)$.
- We consider algorithms which are *polynomial time in* $A$. This means algorithms which run in polynomial time in the length of the input and which, besides the usual computational steps, can also consult a "black box" for solving problem $A(y)$ for any intermediate output $y$.
- We s me in $A$, i.e. orithm whic
- Note that we do NOT require that $A$ be an NP problem; we even do not require it to be a decision problem - it can also be an optimisa

# NP hard problems

- Let $A$ be a problem and assume that we have a 'black box" device (also called "an oracle") which for every input $x$ instantaneously computes $A(x)$.
- We consider algorithms which are *polynomial time in* $A$. This means algorithms which run in polynomial time in the length of the input and which, besides the usual computational steps, can also consult a "black box" for solving problem $A(y)$ for any intermediate output $y$.
- We s me in $A$, i.e. orithm whic
- Note that we do NOT require that $A$ be an NP problem; we even do not require it to be a decision problem - it can also be an optimisa
- Example: (The Traveling Salesman Optimisatio

# NP hard problems

- Let $A$ be a problem and assume that we have a 'black box" device (also called "an oracle") which for every input $x$ instantaneously computes $A(x)$.
- We consider algorithms which are *polynomial time in* $A$. This means algorithms which run in polynomial time in the length of the input and which, besides the usual computational steps, can also consult a "black box" for solving problem $A(y)$ for any intermediate output $y$.
- We s me in $A$, i.e. orithm whic
- Note that we do NOT require that $A$ be an NP problem; we even do not require it to be a decision problem - it can also be an optimisa
- Example: (The Traveling Salesman Optimisatio
  - Instance: A weighted graph (a map of locations representing the lengths of the edges of the grap locations);

# NP hard problems

- Let $A$ be a problem and assume that we have a 'black box" device (also called "an oracle") which for every input $x$ instantaneously computes $A(x)$.
- We consider algorithms which are *polynomial time in* $A$. This means algorithms which run in polynomial time in the length of the input and which, besides the usual computational steps, can also consult a "black box" for solving problem $A(y)$ for any intermediate output $y$.
- We s me in $A$, i.e. orithm whic
- Note that we do NOT require that $A$ be an NP problem; we even do not require it to be a decision problem - it can also be an optimisa
- Example: (The Traveling Salesman Optimisatio
  - Instance: A weighted graph (a map of locations representing the lengths of the edges of the grap locations);
  - Problem: Find a cycle in the graph containing all vertices which is of minimal possible total length.

# NP hard problems

- Let $A$ be a problem and assume that we have a 'black box" device (also called "an oracle") which for every input $x$ instantaneously computes $A(x)$.
- We consider algorithms which are *polynomial time in* $A$. This means algorithms which run in polynomial time in the length of the input and which, besides the usual computational steps, can also consult a "black box" for solving problem $A(y)$ for any intermediate output $y$.
- We s me in $A$, i.e. orithm whic
- Note that we do NOT require that $A$ be an NP problem; we even do not require it to be a decision problem - it can also be an optimisa
- Example: (The Traveling Salesman Optimisatio
  - Instance: A weighted graph (a map of locations representing the lengths of the edges of the grap locations);
  - Problem: Find a cycle in the graph containing all vertices which is of minimal possible total length.
- Think of a mailman having to deliver mail to several addresses while having to travel as small total distance as possible.

# NP hard problems



- The Traveling Salesman Optimisation Problem is clearly NP hard:
- using a "black box" for solving it, we can solve the Traveling Salesman Decision problem:
- Giv ycle cont
- We d determining the length of the cycle of minimal possible length and comparing the length of such a cycle with $L$.
- Since all other NP problems are polynomial time redu Salesman Decision problem (which is NP complete) problem is solvable using a "black box" for the Traveling Salesman Optimisation Problem.

# The significance of NP hard problems

- It is important to be able to figure out if a problem at hand is NP hard in order to know that one has to abandon trying to come up with a feasible (i.e., polynomial time) solution.

# The significance of NP hard problems

- It is important to be able to figure out if a problem at hand is NP hard in order to know that one has to abandon trying to come up with a feasible (i.e., polynomial time) solution.
- So what do we do when we encounter an NP hard problem?

# The significance of NP hard problems

- It is important to be able to figure out if a problem at hand is NP hard in order to know that one has to abandon trying to come up with a feasible (i.e., polynomial time) solution.
- So what do we do when we encounter an NP hard problem?
- If this
in an a
it is rea

# The significance of NP hard problems

- It is important to be able to figure out if a problem at hand is NP hard in order to know that one has to abandon trying to come up with a feasible (i.e., polynomial time) solution.
- So what do we do when we encounter an NP hard problem?
- If this
in an a
it is rea
- For e
might look for a tour which is not longer than twice the length of the shortest possible tour.

# The significance of NP hard problems

- It is important to be able to figure out if a problem at hand is NP hard in order to know that one has to abandon trying to come up with a feasible (i.e., polynomial time) solution.
- So what do we do when we encounter an NP hard problem?
- If this
in an a
it is rea
- For e
might look for a tour which is not longer than twice the length of the shortest possible tour.
- Thus, for a practical problem which appears to be hard,

# The significance of NP hard problems

- It is important to be able to figure out if a problem at hand is NP hard in order to know that one has to abandon trying to come up with a feasible (i.e., polynomial time) solution.
- So what do we do when we encounter an NP hard problem?
- If this
in an a
it is rea
- For e
might look for a tour which is not longer than twice the length of the shortest possible tour.
- Thus, for a practical problem which appears to be hard,
    - prove that the problem is indeed NP hard, to justify not trying solving the problem exactly;

# The significance of NP hard problems

- It is important to be able to figure out if a problem at hand is NP hard in order to know that one has to abandon trying to come up with a feasible (i.e., polynomial time) solution.
- So what do we do when we encounter an NP hard problem?
- If this
in an a
it is rea
- For e
might look for a tour which is not longer than twice the length of the shortest possible tour.
- Thus, for a practical problem which appears to be hard,
  - prove that the problem is indeed NP hard, to justify not trying solving the problem exactly;
  - look for an approximation algorithm which provides a feasible sub-optimal solution that it is not too bad.

# Proving NP completeness

Warning: sometimes distinction between a problem in P and an NP complete problem can be subtle!

| in P | NP complete |
| --- | --- |
| • Given a graph $G$ and two vertices $s$ an of len | • Given a graph $G$ and two vertices ? |
| • G CNF form such that every clause has at most **two** propositional variables, does the formula have a satisfying assignment? | a in CNF form such that every clause has at m nal vari sati |
| • Given a graph $G$, does $G$ have a tour where every **edge** is traversed exactly once? (An *Euler tour*.) | • Given a graph $G$, does $G$ have a tour where every **vertex** is visited exactly once? (A *Hamiltonian* cycle.) |

# Proving NP completeness

Taking for granted that SAT is NP complete, how do we prove NP completeness of another NP problem??

# Proving NP completeness

Taking for granted that SAT is NP complete, how do we prove NP completeness of another NP problem??

- **Theorem:** Let $U$ be an NP complete problem, and let $V$ be another NP problem. If $U$ is polynomially reducible to $V$ then $V$ is also NP complete.

# Proving NP completeness

Taking for granted that SAT is NP complete, how do we prove NP completeness of another NP problem??

- **Theorem:** Let $U$ be an NP complete problem, and let $V$ be another NP problem. If $U$ is polynomially reducible to $V$ then $V$ is also NP complete.
- Proof: Assume that $g(x)$ is a polynomial reduction of $U$ to $V$, and let $W$ be any o

# Proving NP completeness

Taking for granted that SAT is NP complete, how do we prove NP completeness of another NP problem??

- **Theorem:** Let $U$ be an NP complete problem, and let $V$ be another NP problem. If $U$ is polynomially reducible to $V$ then $V$ is also NP complete.
- Proof: Assume that $g(x)$ is a polynomial reduction of $U$ to $V$, and let $W$ be any o
- Sinc of $W$ to $U$.

# Proving NP completeness

Taking for granted that SAT is NP complete, how do we prove NP completeness of another NP problem??

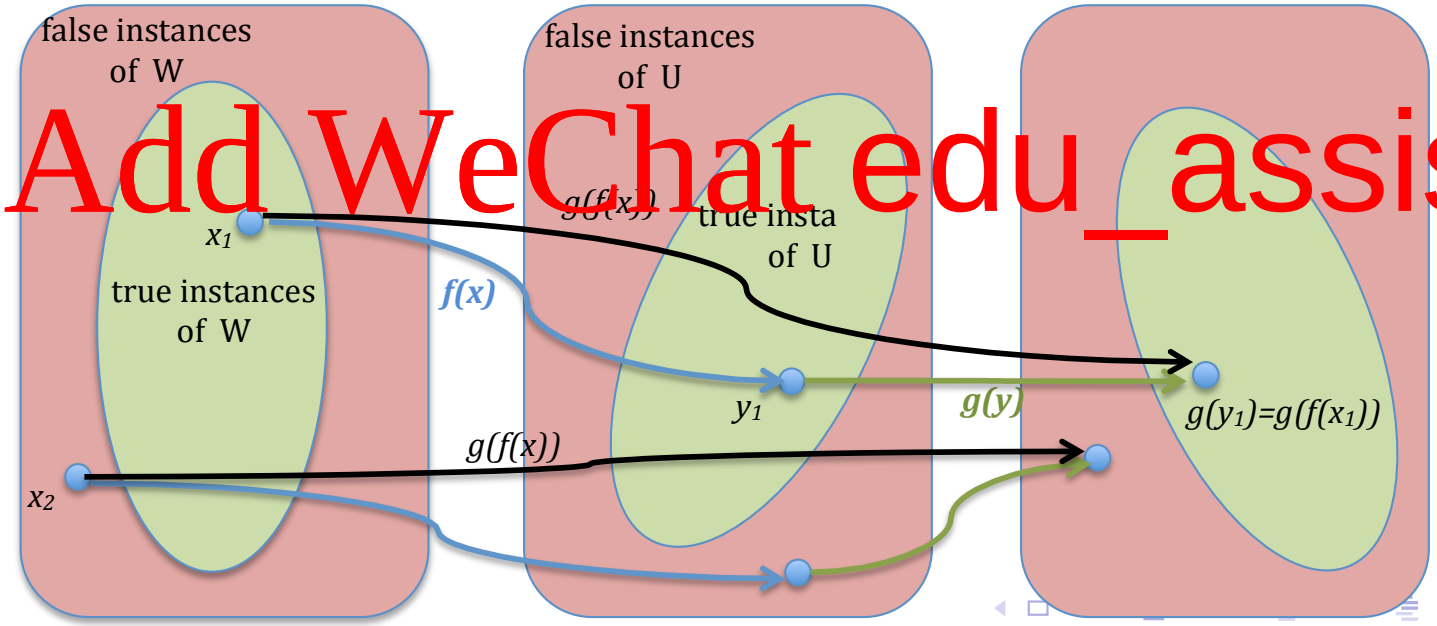
- **Theorem:** Let $U$ be an NP complete problem, and let $V$ be another NP problem. If $U$ is polynomially reducible to $V$ then $V$ is also NP complete.
- Proof: Assume that $g(x)$ is a polynomial reduction of $U$ to $V$, and let $W$ be any o
- Sinc of $W$ to $U$.
- We cl

# Proving NP completeness

Taking for granted that SAT is NP complete, how do we prove NP completeness of another NP problem??

- **Theorem:** Let $U$ be an NP complete problem, and let $V$ be another NP problem. If $U$ is polynomially reducible to $V$ then $V$ is also NP complete.
- Proof: Assume that $g(x)$ is a polynomial reduction of $U$ to $V$, and let $W$ be any o
- Sinc of $W$ to $U$.
- We cl





# Proving NP completeness

- $g(f(x))$ is a polynomial time reduction of $W$ to $V$ because:

# Proving NP completeness

- $g(f(x))$ is a polynomial time reduction of $W$ to $V$ because:
  1. since $f$ is a reduction of $W$ to $U$, $W(x)$ is true just in case $U(f(x))$ is true;

# Proving NP completeness

- $g(f(x))$ is a polynomial time reduction of $W$ to $V$ because:
  1. since $f$ is a reduction of $W$ to $U$, $W(x)$ is true just in case $U(f(x))$ is true;
  2. since $g$ is a reduction of $U$ to $V$, $U(f(x))$ is true just in case $V(g(f(x)))$ is true.

# Proving NP completeness

- $g(f(x))$ is a polynomial time reduction of $W$ to $V$ because:
  1. since $f$ is a reduction of $W$ to $U$, $W(x)$ is true just in case $U(f(x))$ is true;
  2. since $g$ is a reduction of $U$ to $V$, $U(f(x))$ is true just in case $V(g(f(x)))$ is true.
- $x))$ is a

# Proving NP completeness

- $g(f(x))$ is a polynomial time reduction of $W$ to $V$ because:
  1. since $f$ is a reduction of $W$ to $U$, $W(x)$ is true just in case $U(f(x))$ is true;
  2. since $g$ is a reduction of $U$ to $V$, $U(f(x))$ is true just in case $V(g(f(x)))$ is true.
  - $x))$ is a
  - $|x|$, i.e.,

  $|f(x)| \leq P(|x|)$.

# Proving NP completeness

- $g(f(x))$ is a polynomial time reduction of $W$ to $V$ because:
  1. since $f$ is a reduction of $W$ to $U$, $W(x)$ is true just in case $U(f(x))$ is true;
  2. since $g$ is a reduction of $U$ to $V$, $U(f(x))$ is true just in case $V(g(f(x)))$ is true.
  - $x))$ is a
  - $|x|$, i.e., $|f(x)| \leq P(|x|)$.
  - since $g(y)$ is polynomial time computable as w polynomial $Q$ such that for every input inates after at most $Q(|y|)$ many steps.

# Proving NP completeness

- $g(f(x))$ is a polynomial time reduction of $W$ to $V$ because:
  1. since $f$ is a reduction of $W$ to $U$, $W(x)$ is true just in case $U(f(x))$ is true;
  2. since $g$ is a reduction of $U$ to $V$, $U(f(x))$ is true just in case $V(g(f(x)))$ is true.
  - $x))$ is a
  - $|x|$, i.e., $|f(x)| \leq P(|x|)$.
  - since $g(y)$ is polynomial time computable as w polynomial $Q$ such that for every input inates after at most $Q(|y|)$ many steps.
  - Thus, the computation of $g(f(x))$ terminates in at most $P(|x|)$ many steps (computation of $f(x)$) plus $Q(|f(x)|) \leq Q(P(|x|))$ many steps (computation of $g(y)$ for $y = f(x)$).

# Proving NP completeness

- $g(f(x))$ is a polynomial time reduction of $W$ to $V$ because:
  1. since $f$ is a reduction of $W$ to $U$, $W(x)$ is true just in case $U(f(x))$ is true;
  2. since $g$ is a reduction of $U$ to $V$, $U(f(x))$ is true just in case $V(g(f(x)))$ is true.
  - $x))$ is a
  - $|x|$, i.e., $|f(x)| \leq P(|x|)$.
  - since $g(y)$ is polynomial time computable as w polynomial $Q$ such that for every input inates after at most $Q(|y|)$ many steps.
  - Thus, the computation of $g(f(x))$ terminates in at most $P(|x|)$ many steps (computation of $f(x)$) plus $Q(|f(x)|) \leq Q(P(|x|))$ many steps (computation of $g(y)$ for $y = f(x)$).
  - In total, the computation of $g(f(x)$ terminates in at most $P(|x|) + Q(P(|x|))$ many steps, which is a polynomial bound in $|x|$.

# Reducing 3SAT to VC

- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that $VC$ is NP complete:

# Reducing 3SAT to VC

- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that $VC$ is NP complete:
  1. for each clause $C_i$ draw a triangle with three vertices $v_1^i, v_2^i, v_3^i$ and three edges connecting these vertices;

# Reducing 3SAT to VC

- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that $VC$ is NP complete:
  1. for each clause $C_i$ draw a triangle with three vertices $v_1^i, v_2^i, v_3^i$ and three edges connecting these vertices;
  2. for each propositional variable $P_k$ draw a segment with vertices labeled as $P_k$ and $\neg P_k$;

# Reducing 3SAT to VC

- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that $VC$ is NP complete:
  1. for each clause $C_i$ draw a triangle with three vertices $v_1^i, v_2^i, v_3^i$ and three edges connecting these vertices;
  2. for each propositional variable $P_k$ draw a segment with vertices labeled as $P_k$ and $\neg P_k$;
  3. for each clause $C_i$ connect the three corresponding vertices $v_1^i, v_2^i, v_3^i$ with one $C_i$;

# Reducing 3SAT to VC

- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that $VC$ is NP complete:
  1. for each clause $C_i$ draw a triangle with three vertices $v_1^i, v_2^i, v_3^i$ and three edges connecting these vertices;
  2. for each propositional variable $P_k$ draw a segment with vertices labeled as $P_k$ and $\neg P_k$;
  3. for each clause $C_i$ connect the three corresponding vertices $v_1^i, v_2^i, v_3^i$ with one $C_i$;

# Reducing 3SAT to VC

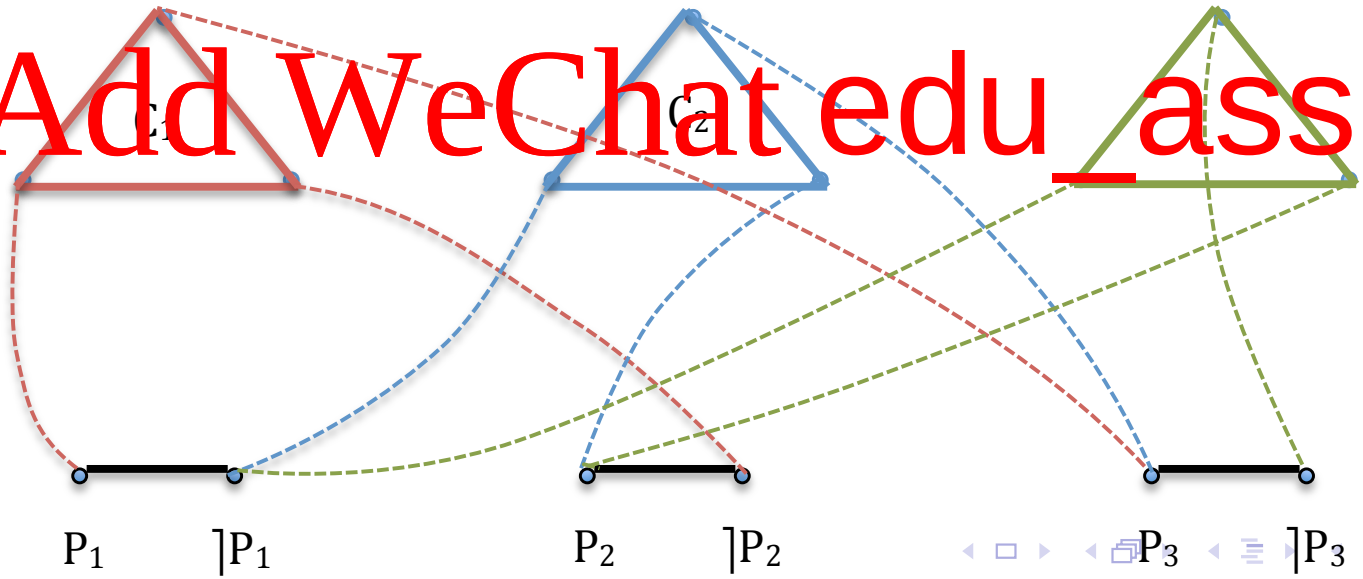
- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that $VC$ is NP complete:
  1. for each clause $C_i$ draw a triangle with three vertices $v_1^i, v_2^i, v_3^i$ and three edges connecting these vertices;
  2. for each propositional variable $P_k$ draw a segment with vertices labeled as $P_k$ and $\neg P_k$;
  3. for each clause $C_i$ connect the three corresponding vertices $v_1^i, v_2^i, v_3^i$ with one $C_i$;

# Reducing 3SAT to VC

- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that $VC$ is NP complete:
  1. for each clause $C_i$ draw a triangle with three vertices $v_1^i, v_2^i, v_3^i$ and three edges connecting these vertices;
  2. for each propositional variable $P_k$ draw a segment with vertices labeled as $P_k$ and $\neg P_k$;
  3. for each clause $C_i$ connect the three corresponding vertices $v_1^i, v_2^i, v_3^i$ with one $C_i$;
  4. Example: $(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \quad P_3) \quad (\ P_1 \quad P_2 \quad P_3)$

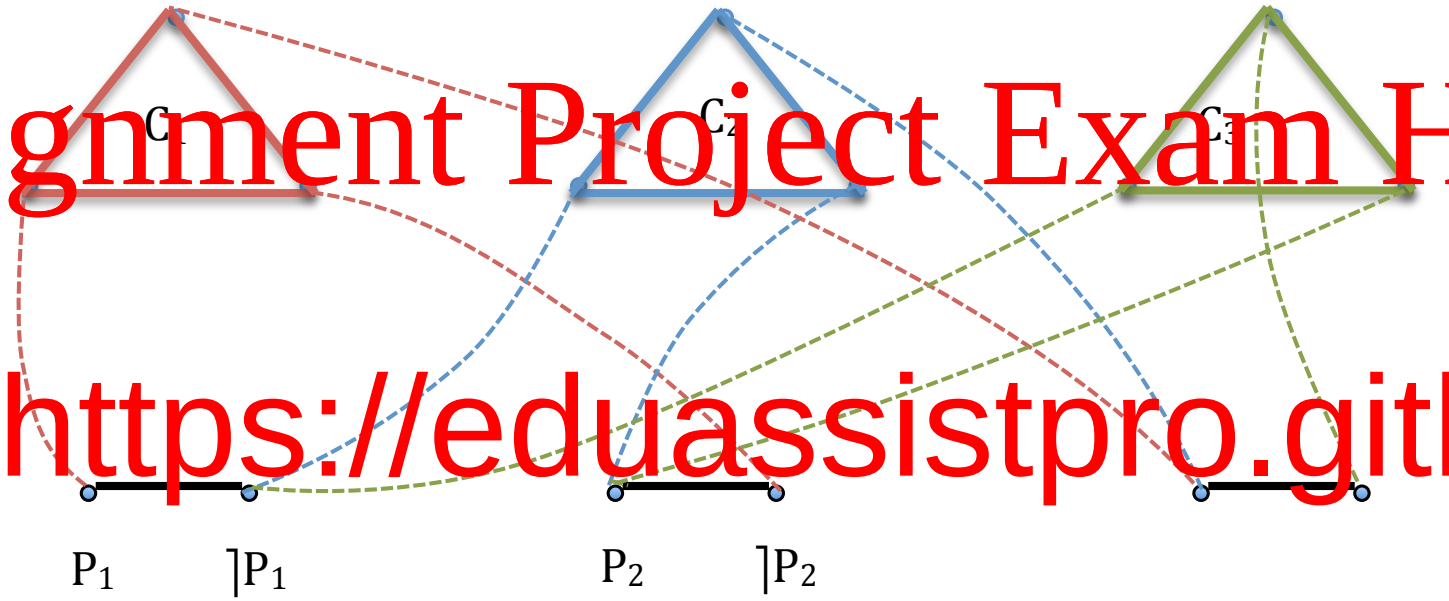
# Reducing 3SAT to VC

- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that $VC$ is NP complete:
  1. for each clause $C_i$ draw a triangle with three vertices $v_1^i, v_2^i, v_3^i$ and three edges connecting these vertices;
  2. for each propositional variable $P_k$ draw a segment with vertices labeled as $P_k$ and $\neg P_k$;
  3. for each clause $C_i$ connect the three corresponding vertices $v_1^i, v_2^i, v_3^i$ with one $C_i$;
  4. Example: $(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \quad P_3) \quad (\quad P_1 \quad P_2 \quad P_3)$



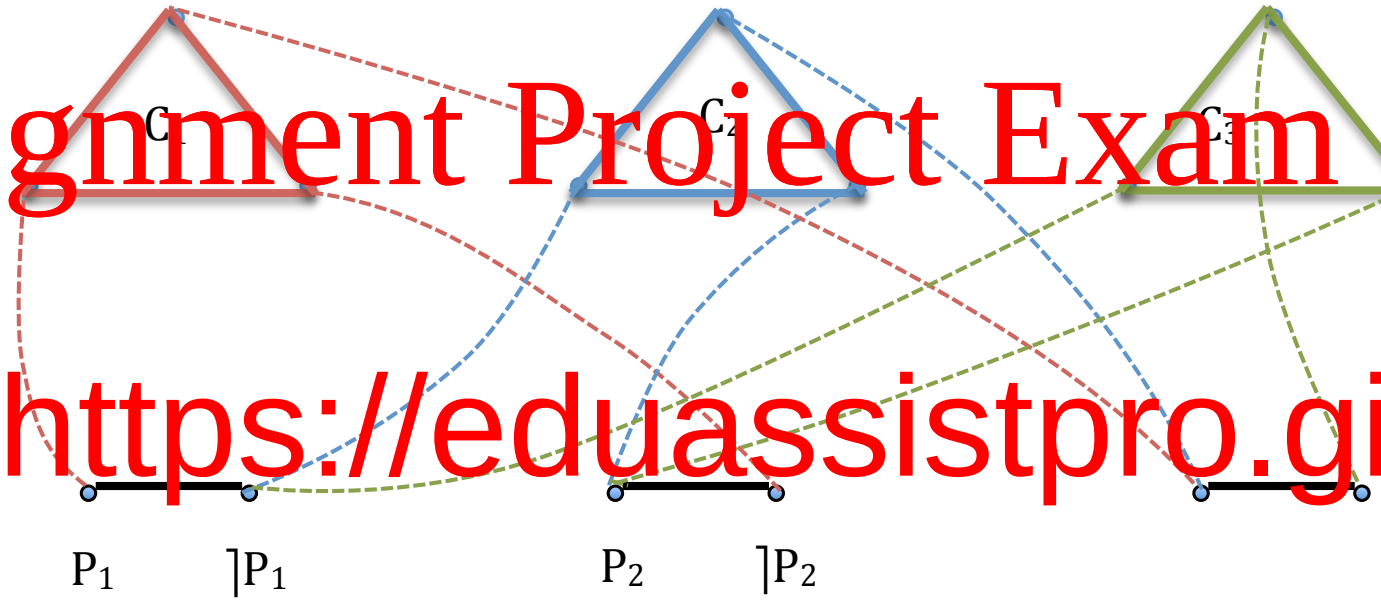
# Reducing 3SAT to VC

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



# Reducing 3SAT to VC

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$

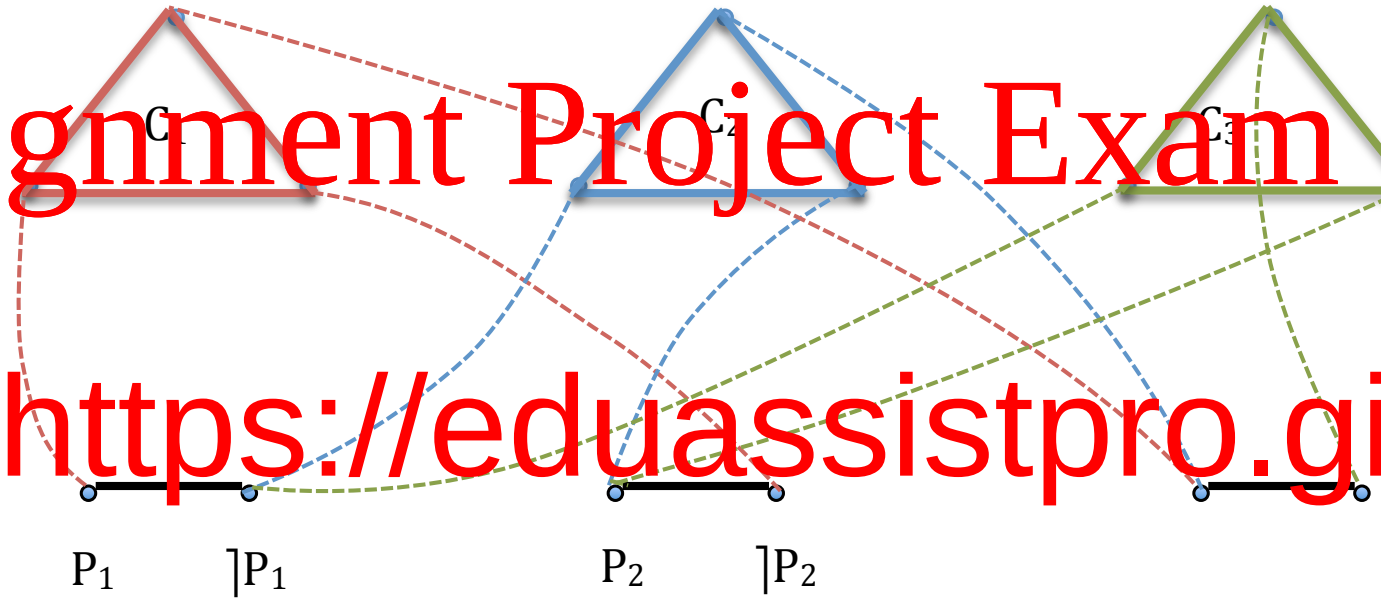





- Claim: an instance of 3SAT consisting of $M$ c $N$ propositional variables has an assignment of variables w if and only if the corresponding graph has a Vertex Cover of siz $+ N$.

# Reducing 3SAT to VC

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$

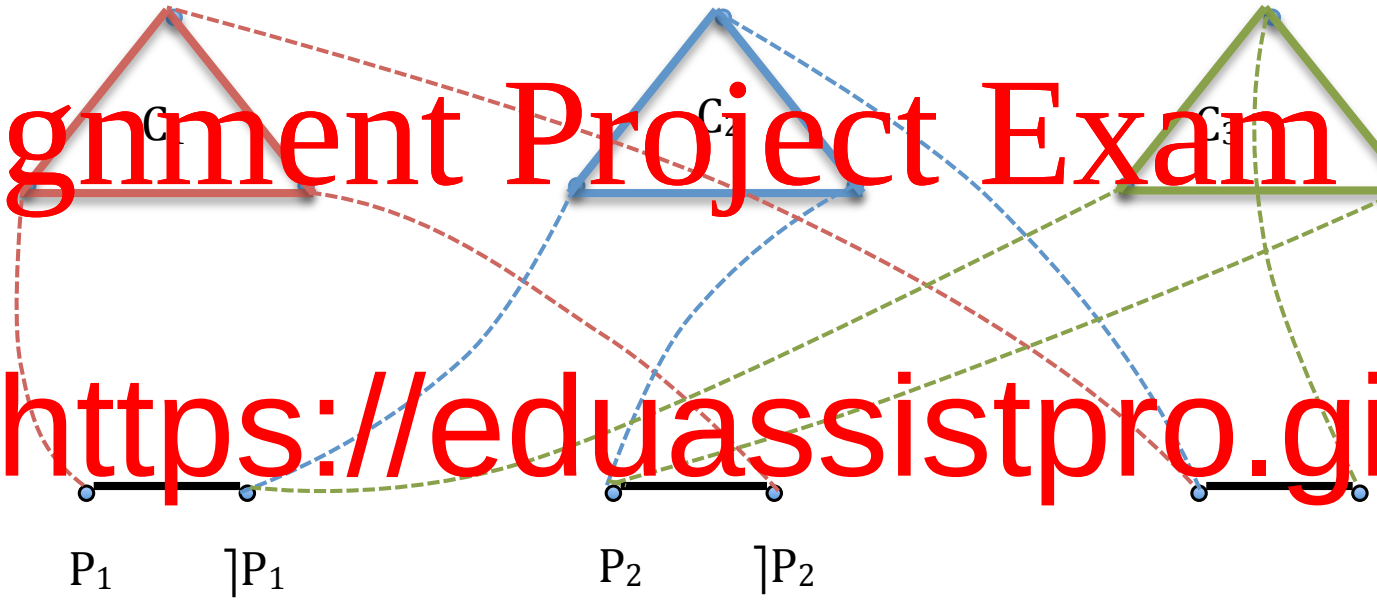





- Claim: an instance of 3SAT consisting of $M$ c V
  propositional variables has an assignment of variables w
  if and only if the corresponding graph has a Vertex Cover of siz $+ N$.
- Assume there is a vertex cover with at most $2M + N$ vertices chosen. Then

# Reducing 3SAT to VC

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$

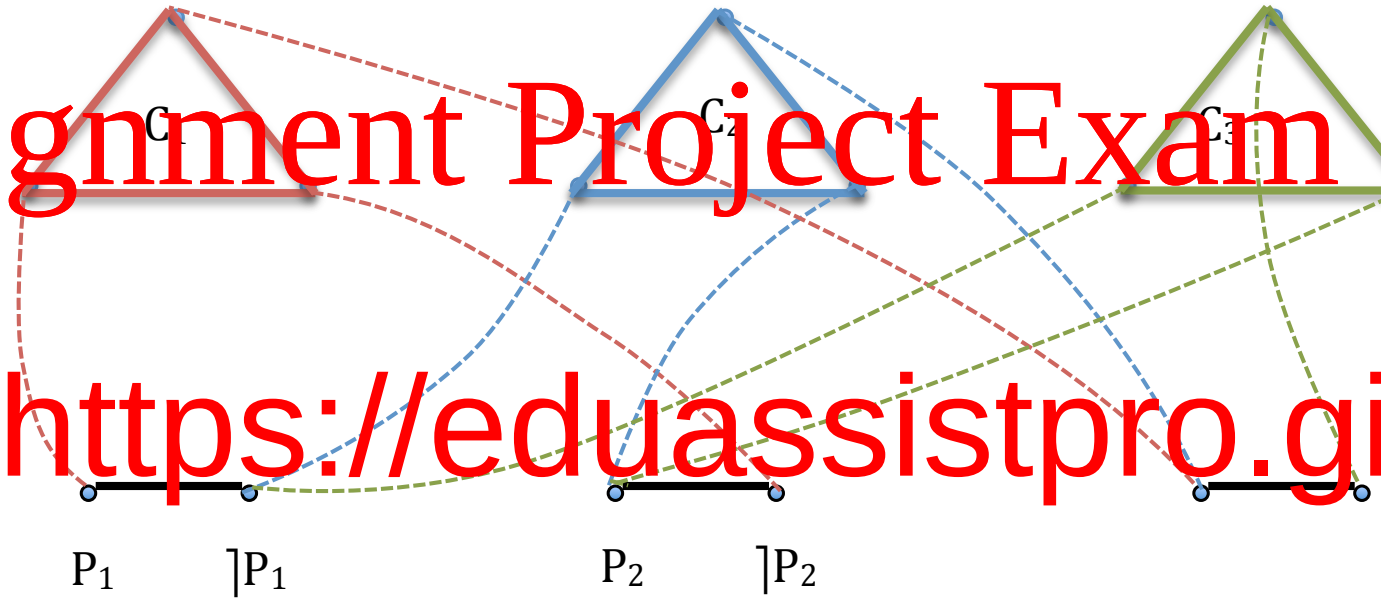





- Claim: an instance of 3SAT consisting of $M$ c $N$ propositional variables has an assignment of variables w if and only if the corresponding graph has a Vertex Cover of siz $+ N$.
- Assume there is a vertex cover with at most $2M + N$ vertices chosen. Then
  1. Each triangle must have at least two vertices chosen;

# Reducing 3SAT to VC

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$

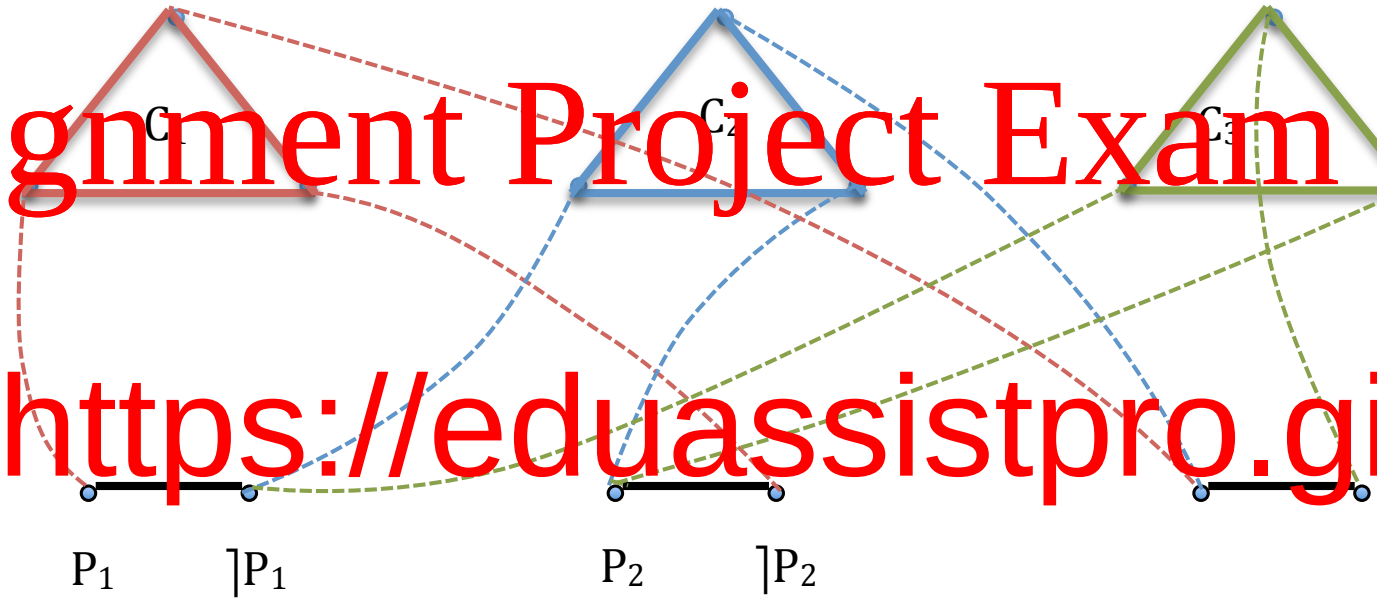





- Claim: an instance of 3SAT consisting of $M$ c $N$ propositional variables has an assignment of variables w if and only if the corresponding graph has a Vertex Cover of siz $+ N$.
- Assume there is a vertex cover with at most $2M + N$ vertices chosen. Then
  1. Each triangle must have at least two vertices chosen;
  2. Each segment must have at least one of its ends chosen.

# Reducing 3SAT to VC

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$

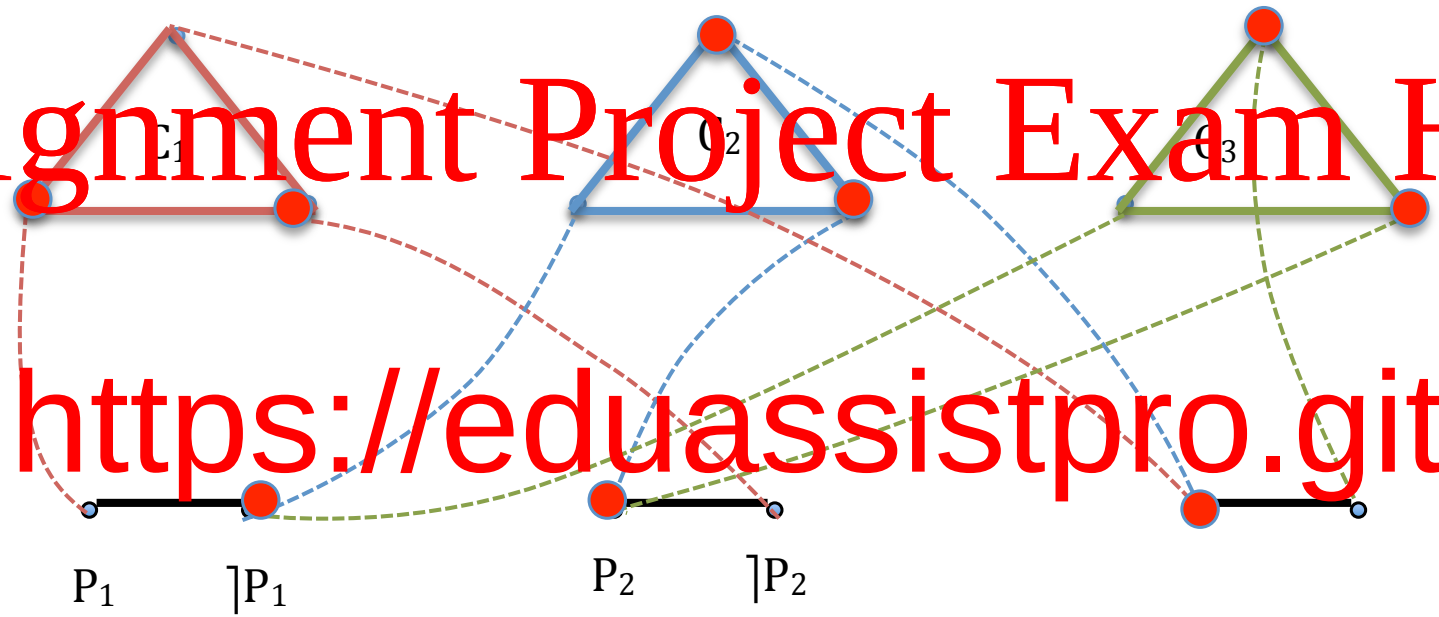





- Claim: an instance of 3SAT consisting of $M$ c
  propositional variables has an assignment of variables w
  if and only if the corresponding graph has a Vertex Cover of siz
  $N$
  $+ N$.
- Assume there is a vertex cover with at most $2M + N$ vertices chosen. Then
  1. Each triangle must have at least two vertices chosen;
  2. Each segment must have at least one of its ends chosen.
- This is in total $2M + N$ points; thus each triangle must have *exactly* two vertices chosen and each segment must have *exactly* one of its ends chosen.

# Reducing 3SAT to VC

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$

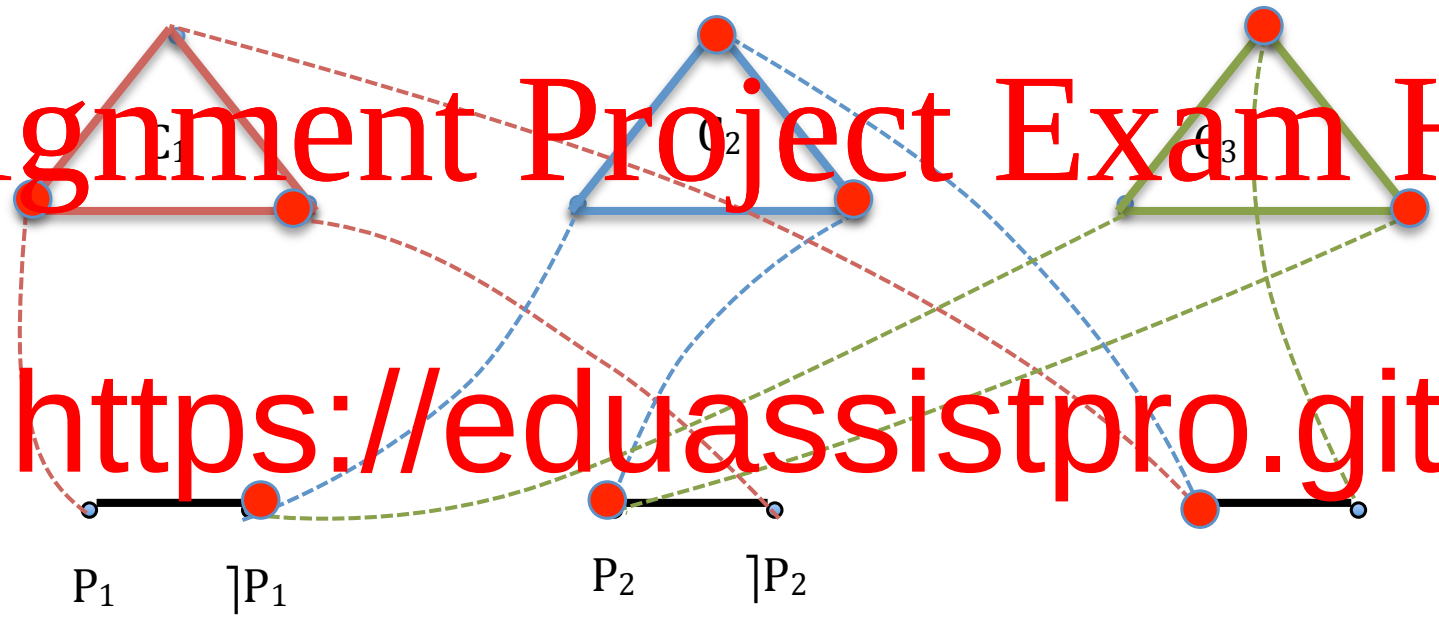





- Set each propositional letter $P_i$ to true if to $P_i$ is covered;

# Reducing 3SAT to VC

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$

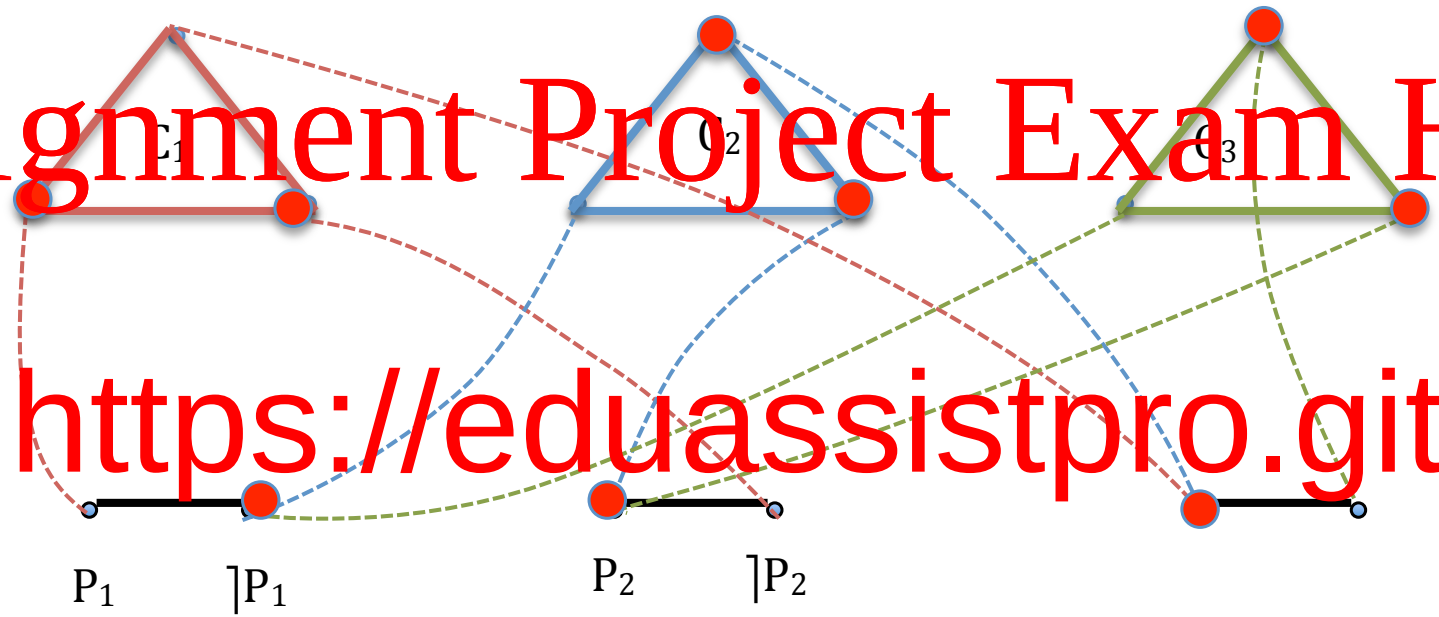





- Set each propositional letter $P_i$ to true if to $P_i$ is covered;
- Otherwise, set a propositional letter $P_i$ to false if $\neg P_i$ is covered,

# Reducing 3SAT to VC

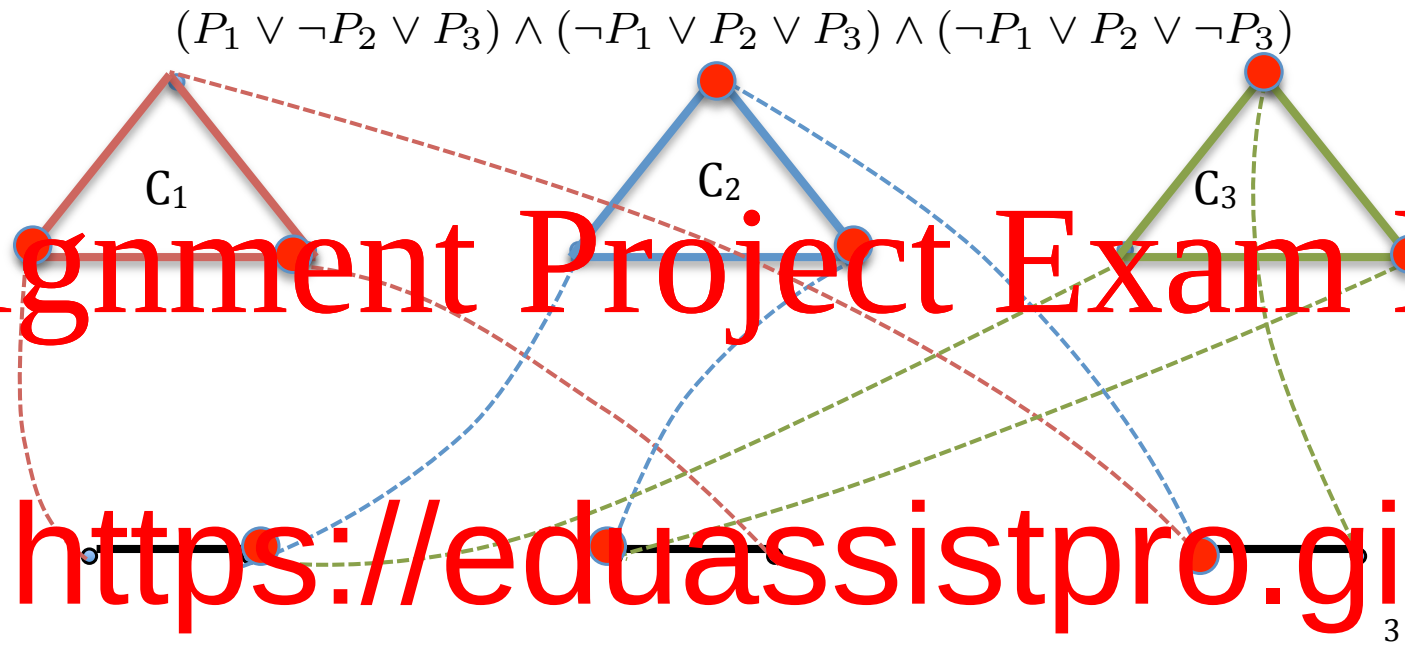
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$






- Set each propositional letter $P_i$ to true if to $P_i$ is covered;
- Otherwise, set a propositional letter $P_i$ to false if $\neg P_i$ is covered,
- In a vertex cover of such a graph every uncovered vertex of each triangle must be connected to a covered end of a segment, which guarantees that the clause corresponding to each triangle is true.

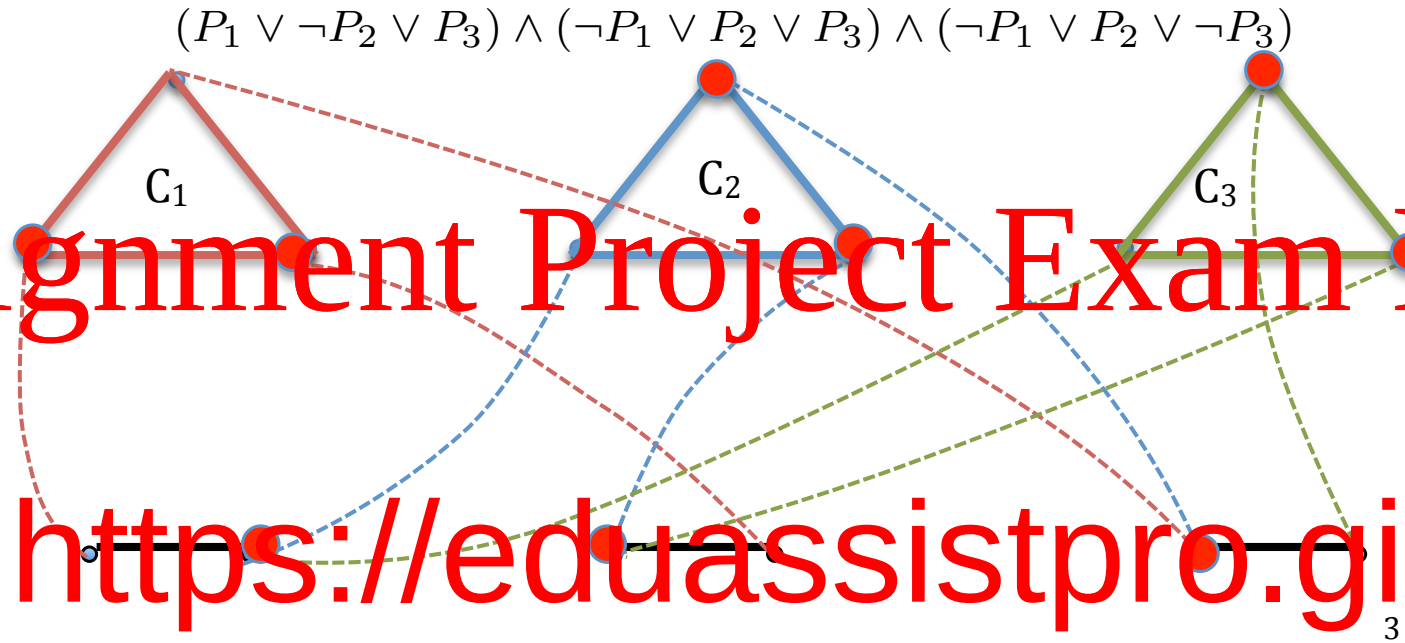
# Reducing 3SAT to VC






- Opposite, assume that the formula has an assignment of th
  true.

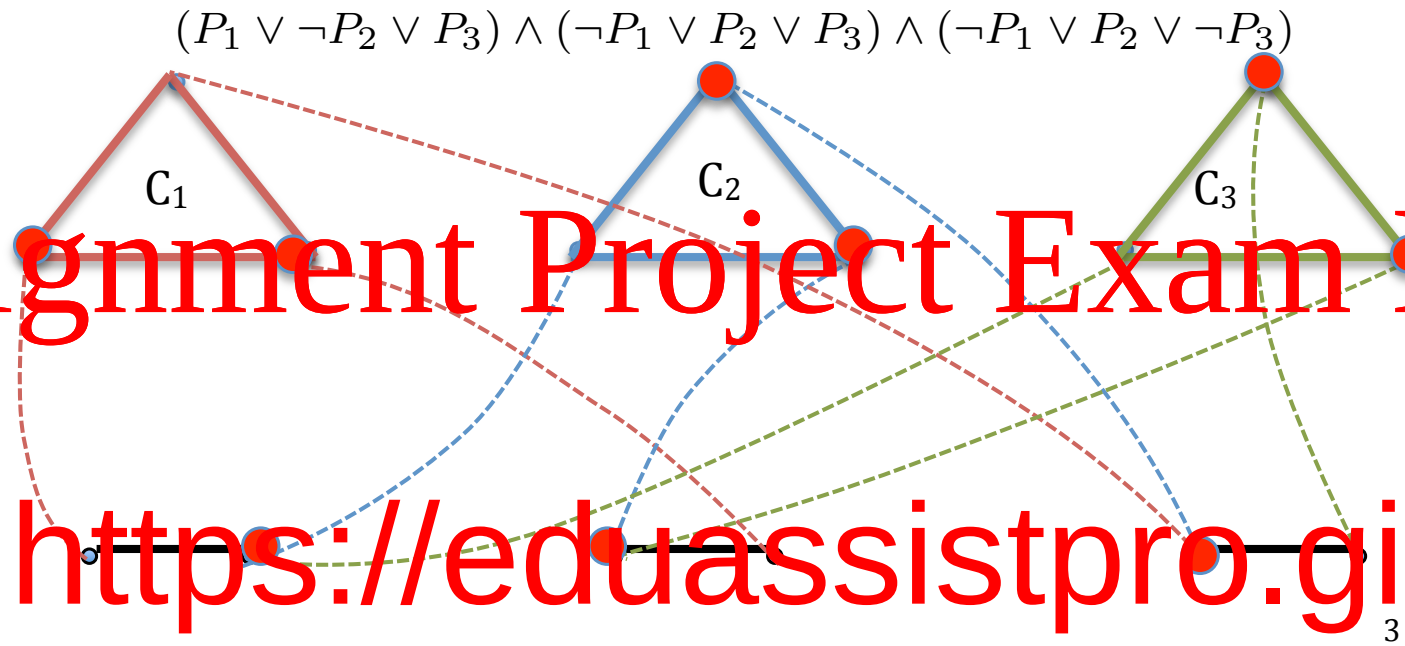
# Reducing 3SAT to VC






- Opposite, assume that the formula has an assignment of th true.
- For each segment, if it corresponds to a propositional letter fying evaluation sets to true cover its $P_i$ end.

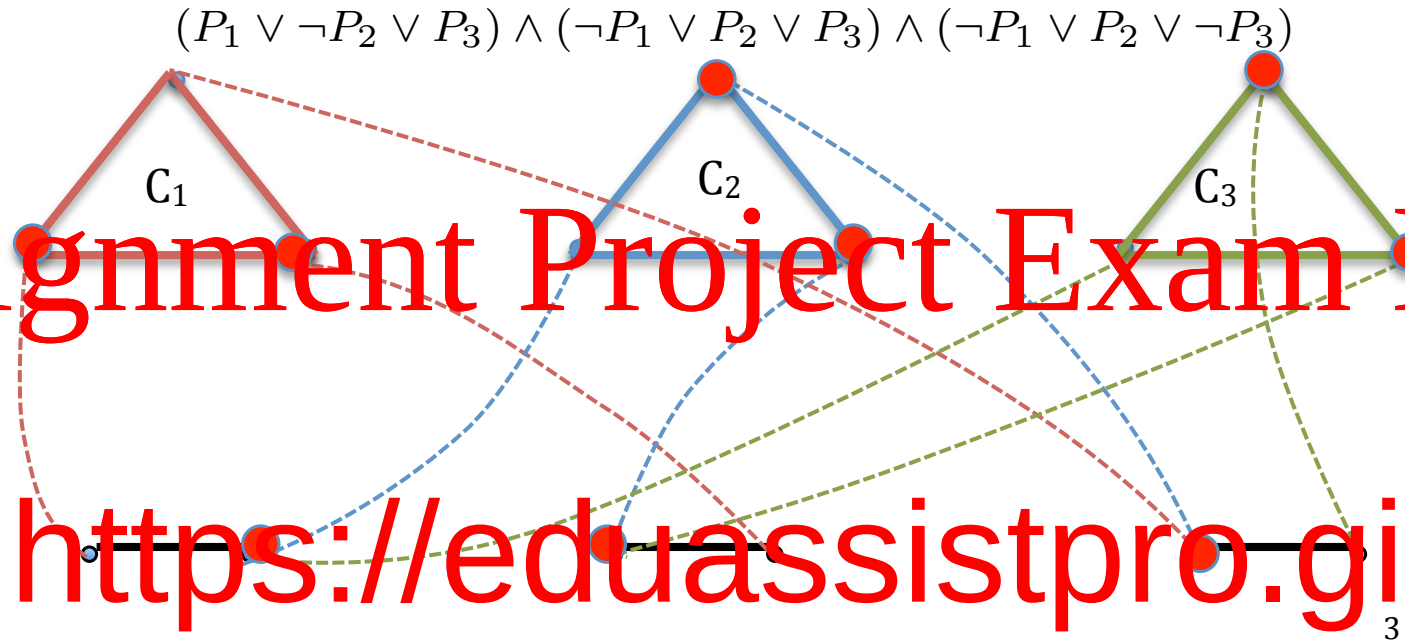
# Reducing 3SAT to VC






- Opposite, assume that the formula has an assignment of th true.
- For each segment, if it corresponds to a propositional letter fying evaluation sets to true cover its $P_i$ end.
- Otherwise, if a propositional letter $P_i$ is set to to false by the satisfying evaluation, cover its $\neg P_i$ end.

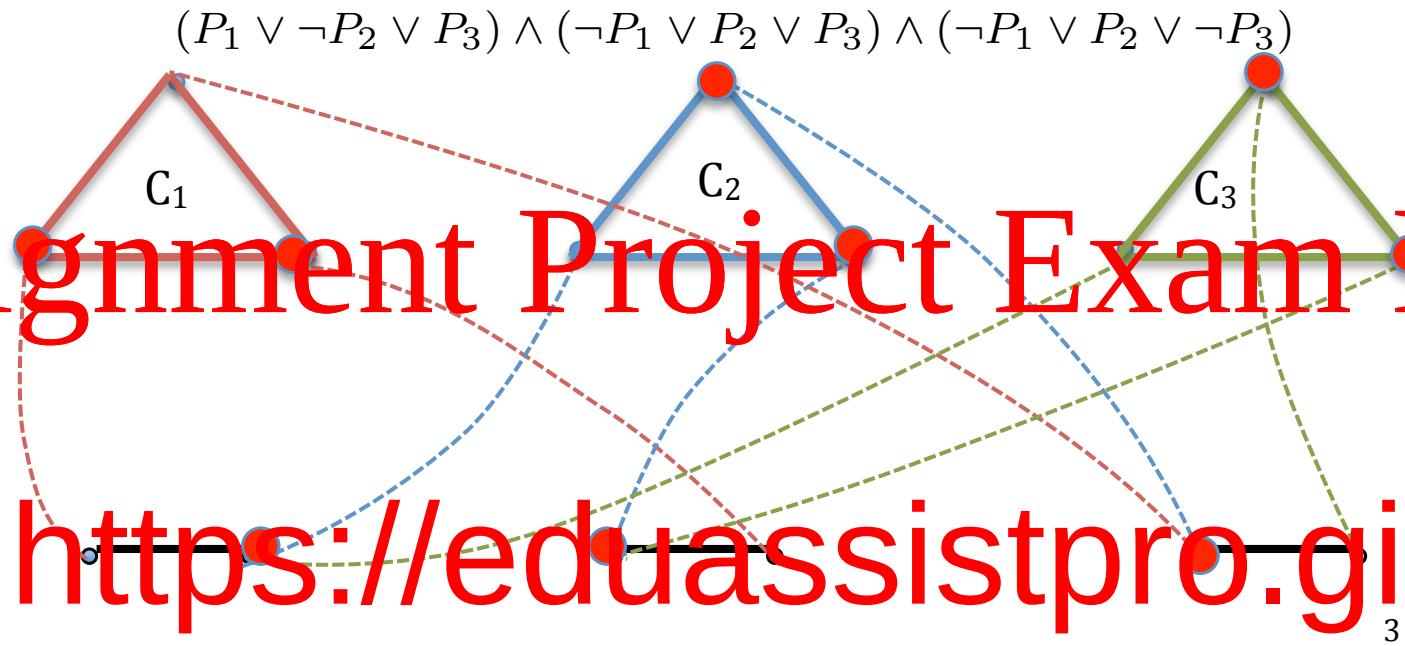
# Reducing 3SAT to VC






- Opposite, assume that the formula has an assignment of th true.
- For each segment, if it corresponds to a propositional letter fying evaluation sets to true cover its $P_i$ end.
- Otherwise, if a propositional letter $P_i$ is set to to false by the satisfying evaluation, cover its $\neg P_i$ end.
- For each triangle corresponding to a clause at least one vertex must be connected to a covered end of a segment, namely to the segment corresponding to the variable which makes that clause true; cover the remaining two vertices of the triangle.

# Reducing 3SAT to VC






- Opposite, assume that the formula has an assignment of th true.
- For each segment, if it corresponds to a propositional letter fying evaluation sets to true cover its $P_i$ end.
- Otherwise, if a propositional letter $P_i$ is set to to false by the satisfying evaluation, cover its $\neg P_i$ end.
- For each triangle corresponding to a clause at least one vertex must be connected to a covered end of a segment, namely to the segment corresponding to the variable which makes that clause true; cover the remaining two vertices of the triangle.
- in this way we cover exactly $2M + N$ vertices of the graph and clearly every edge between a segment and a triangle has at least one end covered.

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.
- Algo

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.
- Algo
  1. 

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.
- Algo
  1. 
  2. 

  covered.

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.
- Algo
  1. 
  2. 
     covered.
  3. Repeat picking edges with both ends uncovere

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.
- Algo
  1. 
  2. 
     covered.
  3. Repeat picking edges with both ends uncovere
- Clearly, this produces a vertex cover because edges ar their end is covered and we perform this procedure unti

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.
- Algo
  1. 
  2. 
     covered.
  3. Repeat picking edges with both ends uncovere
- Clearly, this produces a vertex cover because edges ar their end is covered and we perform this procedure unti
- The number of vertices covered is equal to twice the number of edges with both ends covered. But the minimal vertex cover must cover at least one vertex of each such edge.

# Dealing with NP hard optimisation problems

- If an optimisation problem is NP hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.
- Algo
  1. 
  2. 
     covered.
  3. Repeat picking edges with both ends uncovere
- Clearly, this produces a vertex cover because edges ar their end is covered and we perform this procedure unti
- The number of vertices covered is equal to twice the number of edges with both ends covered. But the minimal vertex cover must cover at least one vertex of each such edge.
- Thus we have produced a vertex cover of size at most twice the size of the minimal vertex cover.

# Dealing with NP hard optimisation problems

- Example: Metric Traveling Salesman Problem (MTSP).

# Dealing with NP hard optimisation problems

- Example: Metric Traveling Salesman Problem (MTSP).
- Instance: A complete weighted graph $G$ with weights $d(i, j)$ of edges (to be interpreted as distances) satisfying the "triangle inequality": for any three vertices $i, j, k$

$$d(i, j) + d(j, k) \geq d(i, k)$$

# Dealing with NP hard optimisation problems

- Example: Metric Traveling Salesman Problem (MTSP).
- Instance: A complete weighted graph $G$ with weights $d(i, j)$ of edges (to be interpreted as distances) satisfying the "triangle inequality": for any three vertices $i, j, k$

$$d(i, j) + d(j, k) \geq d(i, k)$$

- Clai
at mo

$opt$

# Dealing with NP hard optimisation problems

- Example: Metric Traveling Salesman Problem (MTSP).
- Instance: A complete weighted graph $G$ with weights $d(i, j)$ of edges (to be interpreted as distances) satisfying the "triangle inequality": for any three vertices $i, j, k$

$$d(i, j) + d(j, k) \geq d(i, k)$$

- Clai
at mo
- Algo
with one of its edges $e$ removed represents a spanning tree, we have that the total weight of $T$ satisfies $w(T) \leq opt - w($

# Dealing with NP hard optimisation problems

- Example: Metric Traveling Salesman Problem (MTSP).
- Instance: A complete weighted graph $G$ with weights $d(i, j)$ of edges (to be interpreted as distances) satisfying the "triangle inequality": for any three vertices $i, j, k$

$$d(i, j) + d(j, k) \geq d(i, k)$$

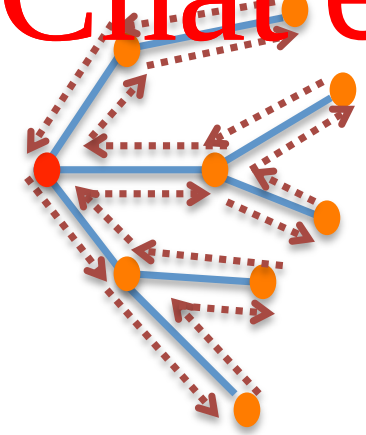
- Clai
at mo
- Algo
with one of its edges $e$ removed represents a spanning tree, we have that the total weight of $T$ satisfies $w(T) \leq opt - w($
- If we do depth-first traverse of the tree, we will travel in tota
$2w(T) \leq 2opt$.

# Dealing with NP hard optimisation problems

- Example: Metric Traveling Salesman Problem (MTSP).
- Instance: A complete weighted graph $G$ with weights $d(i, j)$ of edges (to be interpreted as distances) satisfying the "triangle inequality": for any three vertices $i, j, k$

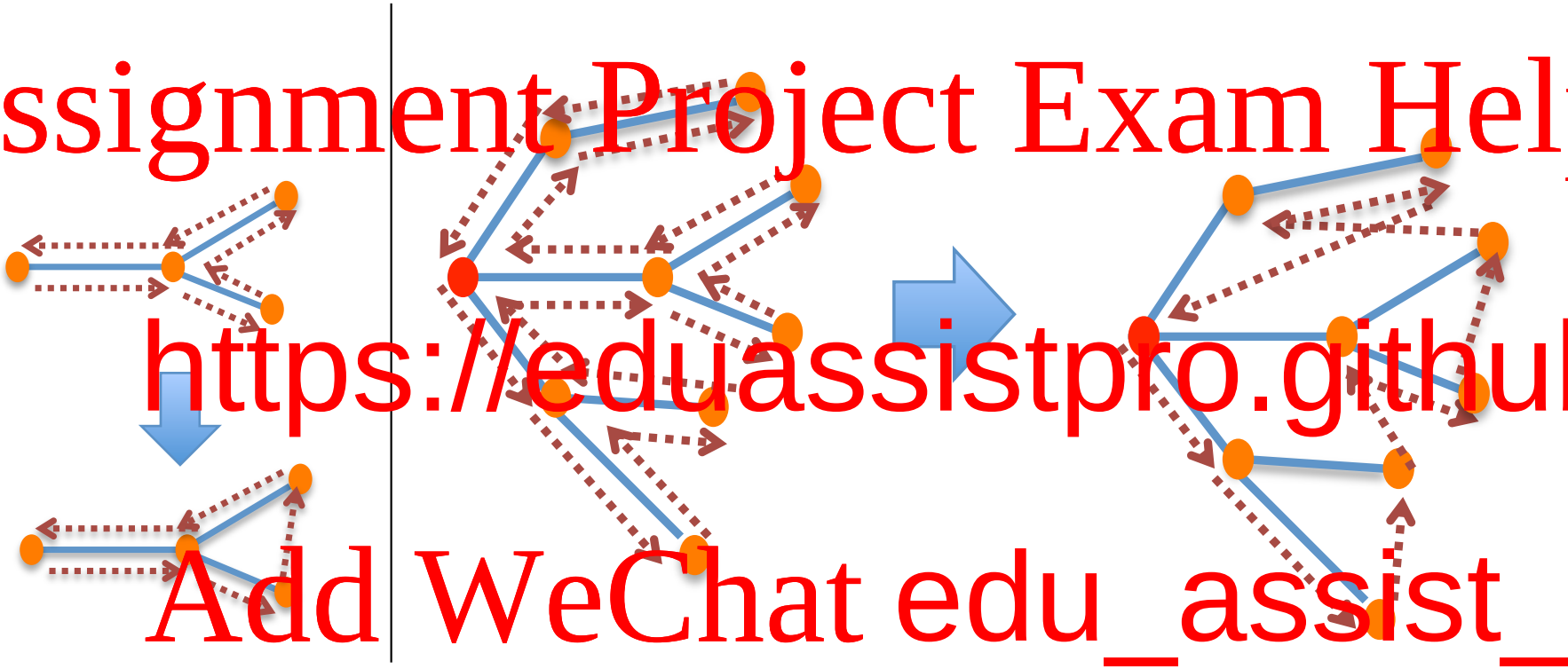
$$d(i, j) + d(j, k) \geq d(i, k)$$

- Clai
at mo
- Algo
with one of its edges $e$ removed represents a spanning tree, we have that the total weight of $T$ satisfies $w(T) \leq opt - w($
- If we do depth-first traverse of the tree, we will travel in tota
$2w(T) \leq 2opt$.



# Dealing with NP hard optimisation problems





- We now take shortcuts to avoid visiting vertices more than once; because of the triangle inequality, this operation does not increase the length of the tour.

# Dealing with NP hard optimisation problems

- As we have mentioned, all NP complete problems are in a sense equally difficult because any of them is reducible to any other via a polynomial time transformation.

# Dealing with NP hard optimisation problems

- As we have mentioned, all NP complete problems are in a sense equally difficult because any of them is reducible to any other via a polynomial time transformation.
- However, in a sense they can also be extremely different: for example, as we have seen, the Vertex Cover problem allows an approximation which produces a cov
size.

# Dealing with NP hard optimisation problems

- As we have mentioned, all NP complete problems are in a sense equally difficult because any of them is reducible to any other via a polynomial time transformation.
- However, in a sense they can also be extremely different: for example, as we have seen, the Vertex Cover problem allows an approximation which produces a cov
size.
- On th
allo / 0 there can be a polynomial time algorithm which for every instan is at most $K$ times longer than the optimal tour of matter how large $K$ is chosen!

# Dealing with NP hard optimisation problems

- As we have mentioned, all NP complete problems are in a sense equally difficult because any of them is reducible to any other via a polynomial time transformation.
- However, in a sense they can also be extremely different: for example, as we have seen, the Vertex Cover problem allows an approximation which produces a cov
size.
- On th / 0 there can allo be a polynomial time algorithm which for every instan is at most $K$ times longer than the optimal tour of matter how large $K$ is chosen!
- To prove this, we show that if there existed algorithm producing a tour which is at most $K$ times longer than the optimal tour, then we could obtain an algorithm which solves in polynomial time the Hamiltonian Cycle Problem, i.e., which for every graph $G$ determines if $G$ contains a cycle visiting all vertices exactly once, which is impossible because this problem is known to be NP complete.

- To see this, let $G$ be an arbitrary graph with $n$ vertices.

# Dealing with NP hard optimisation problems

- To see this, let $G$ be an arbitrary graph with $n$ vertices.
- We turn this graph into a complete weighted graph $G^*$ by setting the weights of all existing edges to 1 and then add edges between the remaining pairs of vertices and set their weights to $K \cdot n$.

# Dealing with NP hard optimisation problems

- To see this, let $G$ be an arbitrary graph with $n$ vertices.
- We turn this graph into a complete weighted graph $G^*$ by setting the weights of all existing edges to 1 and then add edges between the remaining pairs of vertices and set their weights to $K \cdot n$.
- We now apply the approximation algorithm (which we have assumed to exist) to produce a tour of all vertices of total length at most $K \ opt$ where $opt$ is the leng

# Dealing with NP hard optimisation problems

- To see this, let $G$ be an arbitrary graph with $n$ vertices.
- We turn this graph into a complete weighted graph $G^*$ by setting the weights of all existing edges to 1 and then add edges between the remaining pairs of vertices and set their weights to $K \cdot n$.
- We now apply the approximation algorithm (which we have assumed to exist) to produce a tour of all vertices of total length at most $K \ opt$ where $opt$ is the leng
- Clea has a tour cons leng

# Dealing with NP hard optimisation problems

- To see this, let $G$ be an arbitrary graph with $n$ vertices.
- We turn this graph into a complete weighted graph $G^*$ by setting the weights of all existing edges to 1 and then add edges between the remaining pairs of vertices and set their weights to $K \cdot n$.
- We now apply the approximation algorithm (which we have assumed to exist) to produce a tour of all vertices of total length at most $K \ \ opt$ where $opt$ is the leng
- Clea has a tour cons leng
- Otherwise, if the original graph $G$ does not h optimal tour through $G^*$ must contain at least o $K \cdot n$

# Dealing with NP hard optimisation problems

- To see this, let $G$ be an arbitrary graph with $n$ vertices.
- We turn this graph into a complete weighted graph $G^*$ by setting the weights of all existing edges to 1 and then add edges between the remaining pairs of vertices and set their weights to $K \cdot n$.
- We now apply the approximation algorithm (which we have assumed to exist) to produce a tour of all vertices of total length at most $K \ \ opt$ where $opt$ is the leng
- Clea has a tour cons leng
- Otherwise, if the original graph $G$ does not h optimal tour through $G^*$ must contain at least o $K \cdot n$
- In this case the optimal tour through $G^*$ $(K \cdot n) + (n-1) \cdot 1 > K \cdot n$.

# Dealing with NP hard optimisation problems

- To see this, let $G$ be an arbitrary graph with $n$ vertices.
- We turn this graph into a complete weighted graph $G^*$ by setting the weights of all existing edges to 1 and then add edges between the remaining pairs of vertices and set their weights to $K \cdot n$.
- We now apply the approximation algorithm (which we have assumed to exist) to produce a tour of all vertices of total length at most $K \ \ opt$ where $opt$ is the leng
- Clea has a tour cons leng
- Otherwise, if the original graph $G$ does not h optimal tour through $G^*$ must contain at least o $K \cdot n$
- In this case the optimal tour through $G^*$ $(K \cdot n) + (n-1) \cdot 1 > K \cdot n$.
- Thus, our approximation algorithm can return a tour of length at most $K \cdot n$ if an only if it actually returns a tour of length of size $n$, which happens just in case $G$ has a Hamiltonian cycle.

# Dealing with NP hard optimisation problems

- To see this, let $G$ be an arbitrary graph with $n$ vertices.
- We turn this graph into a complete weighted graph $G^*$ by setting the weights of all existing edges to 1 and then add edges between the remaining pairs of vertices and set their weights to $K \cdot n$.
- We now apply the approximation algorithm (which we have assumed to exist) to produce a tour of all vertices of total length at most $K \ \ opt$ where $opt$ is the leng
- Clea has a tour cons leng
- Otherwise, if the original graph $G$ does not h optimal tour through $G^*$ must contain at least o $K \cdot n$
- In this case the optimal tour through $G^*$ $(K \cdot n) + (n-1) \cdot 1 > K \cdot n$.
- Thus, our approximation algorithm can return a tour of length at most $K \cdot n$ if an only if it actually returns a tour of length of size $n$, which happens just in case $G$ has a Hamiltonian cycle.
- This is impossible, because this would be a polynomial time decision procedure for determining in $G$ has a Hamiltonian cycle.