Algorithms: COMP3121/9101

Aleks Ignjatović

School of Computer Science and Engineering
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 *decision problem* is a problem with a YES/NO answer.

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 *decision problem* is a problem with a YES/NO answer.

- Examples are:
  - "*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 *decision problem* is a problem with a YES/NO answer.

- Examples are:
  - "*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 *decision problem* is a problem with a YES/NO answer.

- Examples are:
  - "*Input number n is a prime number.*"
  - "*Input graph G is connected.*"
  - "*Input graph G has a cycle containing all vertices of G.*"

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 *decision problem* is a problem with a YES/NO answer.

- Examples are:
  - "*Input number n is a prime number.*"
  - "*Input graph G is connected.*"
  - "*Input graph G has a cycle containing all vertices of G.*"

- We say that a decision problem A is in *polynomial time* if there exists a polynomial time algorithm which solves it.

- 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 *decision problem* is a problem with a YES/NO answer.
- Examples are:
  - "*Input number n is a prime number.*"
  - "*Input graph G is connected.*"
  - "*Input graph G has a cycle containing all vertices of G.*"

- 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 *decision problem* is a problem with a YES/NO answer.
- Examples are:
  - "*Input number n is a prime number.*"
  - "*Input graph G is connected.*"
  - "*Input graph G has a cycle containing all vertices of G.*"

- 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 $A \in \mathbf{P}$.

- What is the *length* of an input?

- 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.
- Examples:

- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.
- Examples:
  - If input x is an integer, then $|x|$ can be taken to be the number of bits in the binary representation of x.

Feasibility of Algorithms

- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.
- Examples:
  - If input x is an integer, then $|x|$ can be taken to be the number of bits in the binary representation of x.
  - 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.
- Examples:
  - If input x is an integer, then $|x|$ can be taken to be the number of bits in the binary representation of x.
  - 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 integer x as the number of digits in the decimal representation of x.

- What is the *length* of an input?
- It is the number of symbols needed to describe the input precisely.
- Examples:
  - If input x is an integer, then $|x|$ can be taken to be the number of bits in the binary representation of x.
  - 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 integer x as the number of digits in the decimal representation of x.
  - This can only change the constants involved in the expression $T(n) = O(n^k)$ but not the asymptotic bound.

- 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.

- 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.

- Alternatively, we can represent G with its adjacency matrix.

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.

- Alternatively, we can represent G with its adjacency matrix.
- If the input graphs are all sparse, this can unnecessarily increase the length of 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.

- Alternatively, we can represent G with its adjacency matrix.
- If the input graphs are all sparse, this can unnecessarily increase the length of the representation of the graph.

- However, since we are interested only in whether the algorithm runs in polynomial time and not in the particular degree of the polynomial bounding 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.
- Alternatively, we can represent G with its adjacency matrix.
- If the input graphs are all sparse, this can unnecessarily increase the length of the representation of the graph.
- However, since we are interested only in whether the algorithm runs in polynomial time and not in the particular degree of the polynomial bounding 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:

- 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

- 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. such that the truth of $B(x, y)$ can be verified by an algorithm running in polynomial time 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. such that the truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x *only*.
- 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. such that the truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x *only*.

- We call y a *certificate* for x.

- For example, decision problem "integer x is not prime" is in $\mathbf{NP}$ because $A(x)$ is true just in case there exists integer y such that $B(x, y) =$"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. such that the truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x *only*.
- We call y a *certificate* for x.
- For example, decision problem "integer x is not prime" is in $\mathbf{NP}$ because $A(x)$ is true just in case there exists integer y such that $B(x, y) =$"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.

- 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. such that the truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x *only*.
- We call y a *certificate* for x.
- For example, decision problem "integer x is not prime" is in $\mathbf{NP}$ because $A(x)$ is true just in case there exists integer y such that $B(x, y) =$"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.

Examples of NP-decision problems:

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 vertex cover of G with at most k elements.

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 vertex cover of G with at most k elements.

- (SAT) Instance: a propositional formula in the CNF form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each clause C_i is a disjunction of propositional variables or their negations, for example

$$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg P_5 \vee \neg P_6) \wedge (\neg P_3 \vee \neg P_4 \vee P_5)$$

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 vertex cover of G with at most k elements.

- (SAT) Instance: a propositional formula in the CNF form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each clause C_i is a disjunction of propositional variables or their negations, for example

$$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg P_5 \vee \neg P_6) \wedge (\neg P_3 \vee \neg P_4 \vee P_5)$$

  Problem: "There exists an evaluation of the propositional variables which makes the formula true".

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 vertex cover of G with at most k elements.

- (SAT) Instance: a propositional formula in the CNF form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each clause C_i is a disjunction of propositional variables or their negations, for example

$$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg P_5 \vee \neg P_6) \wedge (\neg P_3 \vee \neg P_4 \vee P_5)$$

  Problem: "There exists an evaluation of the propositional variables which 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.

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 vertex cover of G with at most k elements.

- (SAT) Instance: a propositional formula in the CNF form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each clause C_i is a disjunction of propositional variables or their negations, for example

$$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg P_5 \vee \neg P_6) \wedge (\neg P_3 \vee \neg P_4 \vee P_5)$$

  Problem: "There exists an evaluation of the propositional variables which 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.

- 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.

- 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 example, is it possible that finding out if one of 2^n possible evaluations of n propositional variables of a propositional formula Φ makes such formula true is not much harder than simply checking if a given particular evaluation of these propositional variables makes Φ 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 example, is it possible that finding out if one of 2^n possible evaluations of n propositional variables of a propositional formula Φ makes such formula true is not much harder than simply checking if a given particular evaluation of these propositional variables makes Φ true?
- Intuitively, this should not be the case; determining if such evaluation exists should be a harder problem than simply checking if a given evaluation makes 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 example, is it possible that finding out if one of 2^n possible evaluations of n propositional variables of a propositional formula Φ makes such formula true is not much harder than simply checking if a given particular evaluation of these propositional variables makes Φ true?
- Intuitively, this should not be the case; determining if such evaluation exists should be a harder problem than simply checking if a given evaluation makes 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 example, is it possible that finding out if one of 2^n possible evaluations of n propositional variables of a propositional formula Φ makes such formula true is not much harder than simply checking if a given particular evaluation of these propositional variables makes Φ true?
- Intuitively, this should not be the case; determining if such evaluation exists should be a harder problem than simply checking if a given evaluation makes 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

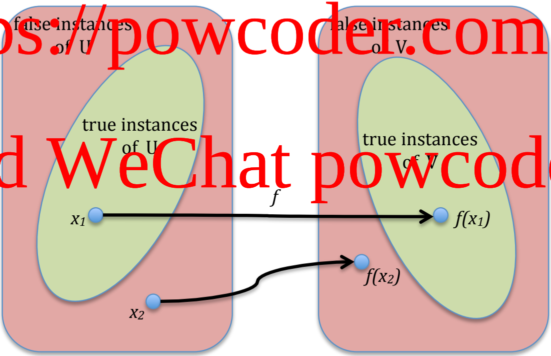
- Let U and V be two decision problems. We say that U is polynomially reducible to V if and only if there exists a function $f(x)$ such that:
  1. $f(x)$ maps instances of U into instances of V;
  2. For every instance x of U we have that $U(x)$ is true just in case $f(x)$ is an instance of V such that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



instances

instances

Example of a polynomial reduction:

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.

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

$$\underbrace{P_1 \vee \neg P_2} \vee \underbrace{\neg P_3} \vee \underbrace{P_4} \vee \underbrace{\neg P_5 \vee P_6} \tag{1}$$

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

$$\underbrace{P_1 \vee \neg P_2} \vee \underbrace{\neg P_3} \vee \underbrace{P_4} \vee \underbrace{\neg P_5 \vee P_6} \tag{1}$$

with the following conjunction of "chained" 3-clauses with new propositional variables Q_1, Q_2, Q_3:

$$(\underbrace{P_1 \vee \neg P_2} \vee Q_1) \wedge (\neg Q_1 \vee \underbrace{\neg P_3} \vee Q_2) \wedge (\neg Q_2 \vee \underbrace{P_4} \vee Q_3) \wedge (\neg Q_3 \vee \underbrace{\neg P_5 \vee P_6}) \tag{2}$$

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

$$\underbrace{P_1 \vee \neg P_2} \vee \underbrace{\neg P_3} \vee \underbrace{P_4} \vee \underbrace{\neg P_5 \vee P_6} \tag{1}$$

with the following conjunction of "chained" 3-clauses with new propositional variables Q_1, Q_2, Q_3:

$$(\underbrace{P_1 \vee \neg P_2} \vee Q_1) \wedge (\neg Q_1 \vee \underbrace{\neg P_3} \vee Q_2) \wedge (\neg Q_2 \vee \underbrace{P_4} \vee Q_3) \wedge (\neg Q_3 \vee \underbrace{\neg P_5 \vee P_6}) \tag{2}$$

- 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

$$P_1 \vee \neg P_2 \vee \neg P_3 \vee P_4 \vee \neg P_5 \vee P_6 \tag{1}$$

with the following conjunction of "chained" 3-clauses with new propositional variables Q_1, Q_2, Q_3:

$$(P_1 \vee \neg P_2 \vee Q_1) \wedge (\neg Q_1 \vee \neg P_3 \vee Q_2) \wedge (\neg Q_2 \vee P_4 \vee Q_3) \wedge (\neg Q_3 \vee \neg P_5 \vee P_6) \tag{2}$$

- 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

$$P_1 \vee \neg P_2 \vee \neg P_3 \vee P_4 \vee \neg P_5 \vee P_6 \tag{1}$$

with the following conjunction of "chained" 3-clauses with new propositional variables Q_1, Q_2, Q_3:

$$(P_1 \vee \neg P_2 \vee Q_1) \wedge (\neg Q_1 \vee \neg P_3 \vee Q_2) \wedge (\neg Q_2 \vee P_4 \vee Q_3) \wedge (\neg Q_3 \vee \neg P_5 \vee P_6) \tag{2}$$

- 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.

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

- **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:

- **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 Φ_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 Φ_x;
  2. $U(x)$ is true just in case $\Phi(x)$ has an evaluation of its propositional variables which makes Φ_x true.

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 Φ_x;
  2. $U(x)$ is true just in case $\Phi(x)$ has an evaluation of its propositional variables which makes Φ_x true.

- An NP decision problem $U(x)$ is NP *complete* if every other NP problem is polynomially reducible to $U(x)$.

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 Φ_x;
  2. $U(x)$ is true just in case $\Phi(x)$ has an evaluation of its propositional variables which makes Φ_x true.

- An NP decision problem $U(x)$ is NP *complete* if every other NP problem is polynomially reducible to $U(x)$.
- 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 Φ_x;
  2. $U(x)$ is true just in case $\Phi(x)$ has an evaluation of its propositional variables which makes Φ_x true.
- An NP decision problem $U(x)$ is NP *complete* if every other NP problem is polynomially reducible to $U(x)$.
- Thus, Cook's Theorem says that SAT is NP complete.
- NP complete problems are in a sense universal: if we had an algorithm that solves one single NP complete problem U, then we could use such an algorithm 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 Φ_x;
  2. $U(x)$ is true just in case $\Phi(x)$ has an evaluation of its propositional variables which makes Φ_x true.
- An NP decision problem $U(x)$ is NP *complete* if every other NP problem is polynomially reducible to $U(x)$.
- Thus, Cook's Theorem says that SAT is NP complete.
- NP complete problems are in a sense universal: if we had an algorithm that solves one single NP complete problem U, then we could use such an algorithm 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 Φ_x;
  2. $U(x)$ is true just in case $\Phi(x)$ has an evaluation of its propositional variables which makes Φ_x true.

- An NP decision problem $U(x)$ is NP *complete* if every other NP problem is polynomially reducible to $U(x)$.

- Thus, Cook's Theorem says that SAT is NP complete.

- NP complete problems are in a sense universal: if we had an algorithm that solves one single NP complete problem U, then we could use such an algorithm 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 Φ_x;
  2. $U(x)$ is true just in case $\Phi(x)$ has an evaluation of its propositional variables which makes Φ_x true.

- An NP decision problem $U(x)$ is NP *complete* if every other NP problem is polynomially reducible to $U(x)$.
- Thus, Cook's Theorem says that SAT is NP complete.
- NP complete problems are in a sense universal: if we had an algorithm that solves one single NP complete problem U, then we could use such an algorithm 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.

- 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 problem, not even an algorithm that runs in time $O(n^{1,000,000})$.

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 problem, not even an algorithm that runs in time $O(n^{1,000,000})$.

- So why bother with them?

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 problem, not even an algorithm that runs in time $O(n^{1,000,000})$.
- So why bother with them?
- Maybe SAT is not that important - why should we care about satisfiability of 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 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 problem, not even an algorithm that runs in time $O(n^{1,000,000})$.

- So why bother with them?

- Maybe SAT is not that important - why should we care about satisfiability of propositional formulas?

- Maybe NP complete problems only have theoretical significance and no 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 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 problem, not even an algorithm that runs in time $O(n^{1,000,000})$.

- So why bother with them?

- Maybe SAT is not that important - why should we care about satisfiability of propositional formulas?

- Maybe NP complete problems only have theoretical significance and no practical relevance??

- Unfortunately, this cannot be farthest from the truth!

- 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 problem, not even an algorithm that runs in time $O(n^{1,000,000})$.

- So why bother with them?
- Maybe SAT is not that important - why should we care about satisfiability of propositional formulas?

- Maybe NP complete problems only have theoretical significance and no practical relevance??
- Unfortunately, this cannot be farthest from the truth!
- A vast number of practically important decision problems are NP complete!

- **Traveling Salesman Problem**

- **Traveling Salesman Problem**
  1. 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 edges connecting these vertices which represent roads connecting these locations and with the weights of these edges representing the lengths of these roads;

NP complete problems are everywhere!

- **Traveling Salesman Problem**

1. Instance:
    1. a map, i.e., a weighted graph with locations as vertices and with edges connecting these vertices which represent roads connecting these locations and with the weights of these edges representing the lengths of these roads;
    2. a number L.

- **Traveling Salesman Problem**

1. Instance:
   1. a map, i.e., a weighted graph with locations as vertices and with edges connecting these vertices which represent roads connecting these locations and with the weights of these edges representing the lengths of these roads;
   2. a number L.
2. Problem: Is there a tour along the edges visiting each location (i.e., vertex) exactly once and returning to the starting location such that the 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 edges connecting these vertices which represent roads connecting these locations and with the weights of these edges representing the lengths of these roads;
    2. a number L.

2. Problem: Is there a tour along the edges visiting each location (i.e., vertex) exactly once and returning to the starting location such that the 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?

- Register Allocation Problem

- **Register Allocation Problem**

1. Instance:

- **Register Allocation Problem**

1. Instance:
    1. A graph G with vertices of the graph representing program variables;

NP complete problems are everywhere!

- **Register Allocation Problem**

1. Instance:
    1. A graph G with vertices of the graph representing program variables;
    2. The edges of the graph indicating that the two variables corresponding to the vertices of that edge are both needed at the same step of the execution of the program;

NP complete problems are everywhere!

- **Register Allocation Problem**

1. Instance:
   1. A graph G with vertices of the graph representing program variables;
   2. The edges of the graph indicating that the two variables corresponding to the vertices of that edge are both needed at the same step of the execution of the program;
   3. The number of registers K of the processor.

NP complete problems are everywhere!

- **Register Allocation Problem**

1. Instance:
   1. A graph G with vertices of the graph representing program variables;
   2. The edges of the graph indicating that the two variables corresponding to the vertices of that edge are both needed at the same step of the execution of the program;
   3. The number of registers K of the processor.
2. Problem: Is it possible to assign variables to registers so that no edge has both vertices assigned to the same register.

NP complete problems are everywhere!

- **Register Allocation Problem**

1. Instance:
    1. A graph G with vertices of the graph representing program variables;
    2. The edges of the graph indicating that the two variables corresponding to the vertices of that edge are both needed at the same step of the execution of the program;
    3. The number of registers K of the processor.
2. Problem: Is it possible to assign variables to registers so that no edge has 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.

- Set Cover Problem

NP complete problems are everywhere!

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

- Set Cover Problem

  - Assume you want to buy DVDs, each with one out of N movie that you like.
  - Unfortunately, the store does not sell individual DVDs but only sells bundles of DVDs; there are M bundles and bundle B_i has price p_i.

- Set Cover Problem
    - Assume you want to buy DVDs, each with one out of N movie that you like.
    - Unfortunately, the store does not sell individual DVDs but only sells bundles of DVDs; there are M bundles and bundle B_i has price p_i.
    - Every movie which you want to buy is in at least one of such bundles; some bundles may contain several movies that you want to buy.
    - For each bundle B_i you have a list l_i of all movies in that bundle.

NP complete problems are everywhere!

- Set Cover Problem
  - Assume you want to buy DVDs, each with one out of N movie that you like.
  - Unfortunately, the store does not sell individual DVDs but only sells bundles of DVDs; there are M bundles and bundle B_i has price p_i.
  - Every movie which you want to buy is in at least one of such bundles; some bundles may contain several movies that you want to buy.
  - For each bundle B_i you have a list l_i of all movies in that bundle.
  - Your goal is to choose a set of bundles which together contain all of the movies you want to buy and which has the smallest total cost.

- **Set CoverProblem**
  - Assume you want to buy DVDs, each with one out of N movie that you like.
  - Unfortunately, the store does not sell individual DVDs but only sells bundles of DVDs; there are M bundles and bundle B_i has price p_i.
  - Every movie which you want to buy is in at least one of such bundles; some bundles may contain several movies that you want to buy.
  - For each bundle B_i you have a list l_i of all movies in that bundle.
  - Your goal is to choose a set of bundles which together contain all of the movies you want to buy and which has the smallest total cost.
- 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 say that a problem A is *NP hard* if every NP problem is polynomial time in A, i.e., if we can solve every NP problem U using a polynomial time algorithm which can also use a black box to solve any instance of A.

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 say that a problem A is *NP hard* if every NP problem is polynomial time in A, i.e., if we can solve every NP problem U using a polynomial time algorithm which can also use a black box to solve any instance of A.
- 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 optimisation problem.

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 say that a problem A is *NP hard* if every NP problem is polynomial time in A, i.e., if we can solve every NP problem U using a polynomial time algorithm which can also use a black box to solve any instance of A.
- 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 optimisation problem.
- Example: (The Traveling Salesman Optimisation Problem)

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 say that a problem A is *NP hard* if every NP problem is polynomial time in A, i.e., if we can solve every NP problem U using a polynomial time algorithm which can also use a black box to solve any instance of A.
- 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 optimisation problem.
- Example: (The Traveling Salesman Optimisation Problem)
  - Instance: A weighted graph (a map of locations) with weights representing the lengths of the edges of the graph (roads between 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 say that a problem A is *NP hard* if every NP problem is polynomial time in A, i.e., if we can solve every NP problem U using a polynomial time algorithm which can also use a black box to solve any instance of A.
- 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 optimisation problem.
- Example: (The Traveling Salesman Optimisation Problem)
  - Instance: A weighted graph (a map of locations) with weights representing the lengths of the edges of the graph (roads between locations);
  - Problem: Find a cycle in the graph containing all vertices which is of minimal possible total length.

- 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 say that a problem A is *NP hard* if every NP problem is polynomial time in A, i.e., if we can solve every NP problem U using a polynomial time algorithm which can also use a black box to solve any instance of A.
- 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 optimisation problem.
- Example: (The Traveling Salesman Optimisation Problem)
  - Instance: A weighted graph (a map of locations) with weights representing the lengths of the edges of the graph (roads between 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:
- Given a weighted graph G and a number L we can determine if there is a cycle containing all vertices of the graph and whose length is at most L.
- We do that by solving the Traveling Salesman Optimisation Problem thus 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 reducible to the Traveling Salesman Decision problem (which is NP complete), then every other NP 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 problem is an optimisation problem, we can try to solve such a problem in an approximate sense by finding a solution which might not be optimal, but it is reasonably close to an optimal 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?
- If this problem is an optimisation problem, we can try to solve such a problem in an approximate sense by finding a solution which might not be optimal, but it is reasonably close to an optimal solution.
- For example, in the case of the Traveling Salesman Optimisation Problem we 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 problem is an optimisation problem, we can try to solve such a problem in an approximate sense by finding a solution which might not be optimal, but it is reasonably close to an optimal solution.

- For example, in the case of the Traveling Salesman Optimisation Problem we 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 strategy would be:

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 problem is an optimisation problem, we can try to solve such a problem in an approximate sense by finding a solution which might not be optimal, but it is reasonably close to an optimal solution.

- For example, in the case of the Traveling Salesman Optimisation Problem we 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 strategy would be:

    - 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 problem is an optimisation problem, we can try to solve such a problem in an approximate sense by finding a solution which might not be optimal, but it is reasonably close to an optimal solution.

- For example, in the case of the Traveling Salesman Optimisation Problem we 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 strategy would be:

  - 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 and t, is there a path from s to t of length **at most** K?	• Given a graph G and two vertices s and t, is there a simple path from s to t of length **at least** K?
• Given a propositional formula in CNF form such that every clause has at most **two** propositional variables, does the formula have a satisfying assignment?	• Given a propositional formula in CNF form such that every clause has at most **three** propositional variables, does the formula have a satisfying assignment?
• 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*.)

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

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 other 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.

- Proof: Assume that $g(x)$ is a polynomial reduction of U to V, and let W be any other NP problem.
- Since U is NP complete, there exists a polynomial reduction $f(x)$ 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??

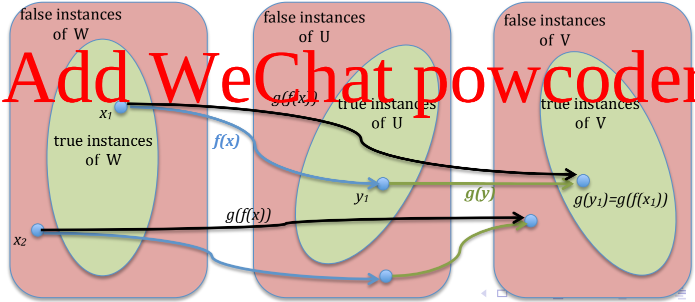
- **Theorem:** Let U be an NP complete problem, and let V be another NP problem. If U is polynomially reducible to V then V is also NP complete.

- Proof: Assume that $g(x)$ is a polynomial reduction of U to V, and let W be any other NP problem.
- Since U is NP complete, there exists a polynomial reduction $f(x)$ of W to U.
- We claim that $g(f(x))$ is a polynomial reduction of W to V.

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 other NP problem.
- Since U is NP complete, there exists a polynomial reduction $f(x)$ of W to U.
- We claim that $g(f(x))$ is a polynomial reduction of W to V.



- $g(f(x))$ 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.

  - Thus $W(x)$ is true just in case $V(g(f(x)))$ is true, i.e., $g(f(x))$ is a reduction of W to V.

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.

  - Thus $W(x)$ is true just in case $V(g(f(x)))$ is true, i.e., $g(f(x))$ is a reduction of W to V.
  - Since $f(x)$ is the output of a polynomial time computable function, the length $|f(x)|$ of the output $f(x)$ can be at most a polynomial in $|x|$, i.e., for some polynomial (with positive coefficients) P we have $|f(x)| \leq P(|x|)$.

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.

  - Thus $W(x)$ is true just in case $V(g(f(x)))$ is true, i.e., $g(f(x))$ is a reduction of W to V.
  - Since $f(x)$ is the output of a polynomial time computable function, the length $|f(x)|$ of the output $f(x)$ can be at most a polynomial in $|x|$, i.e., for some polynomial (with positive coefficients) P we have $|f(x)| \leq P(|x|)$.
  - since $g(y)$ is polynomial time computable as well, there exists a polynomial Q such that for every input y computation of $g(y)$ terminates after at most $Q(|y|)$ many steps.

- $g(f(x))$ 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.

- Thus $W(x)$ is true just in case $V(g(f(x)))$ is true, i.e., $g(f(x))$ is a reduction of W to V.
- Since $f(x)$ is the output of a polynomial time computable function, the length $|f(x)|$ of the output $f(x)$ can be at most a polynomial in $|x|$, i.e., for some polynomial (with positive coefficients) P we have $|f(x)| \leq P(|x|)$.

- since $g(y)$ is polynomial time computable as well, there exists a polynomial Q such that for every input y computation of $g(y)$ terminates 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 $y = f(x)$).

- $g(f(x))$ 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.

  - Thus $W(x)$ is true just in case $V(g(f(x)))$ is true, i.e., $g(f(x))$ is a reduction of W to V.
  - Since $f(x)$ is the output of a polynomial time computable function, the length $|f(x)|$ of the output $f(x)$ can be at most a polynomial in $|x|$, i.e., for some polynomial (with positive coefficients) P we have $|f(x)| \leq P(|x|)$.
  - since $g(y)$ is polynomial time computable as well, there exists a polynomial Q such that for every input y computation of $g(y)$ terminates 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 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 end of the three segments corresponding to the variable appearing in 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 end of the three segments corresponding to the variable appearing in C_i;
     - if the variable appears with a negation sign, connect it with the end labeled with the negation of that letter;

Reducing 3SAT to VC

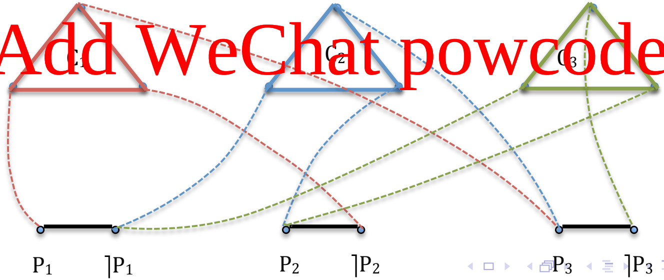
- Reducing an instance of 3SAT to an instance of a Vertex Cover (VC) problem, thus proving that VC is NP complete:
  1. for each clause C_i draw a triangle with three vertices v_1^i, v_2^i, v_3^i and three edges connecting these vertices;
  2. for each propositional variable P_k draw a segment with vertices 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 end of the three segments corresponding to the variable appearing in C_i;
     - if the variable appears with a negation sign, connect it with the end labeled with the negation of that letter;
     - otherwise connect it with the end labeled with that letter;

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 end of the three segments corresponding to the variable appearing in C_i;
     - if the variable appears with a negation sign, connect it with the end labeled with the negation of that letter;
     - otherwise connect it with the end labeled with that letter;
  4. Example: $(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$

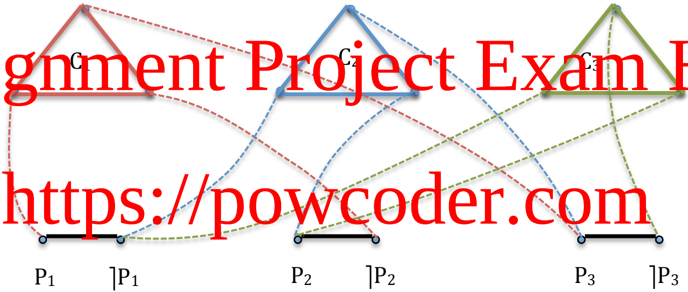
Reducing 3SAT to VC

- 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 end of the three segments corresponding to the variable appearing in C_i;
     - if the variable appears with a negation sign, connect it with the end labeled with the negation of that letter;
     - otherwise connect it with the end labeled with that letter;
  4. Example: $(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$



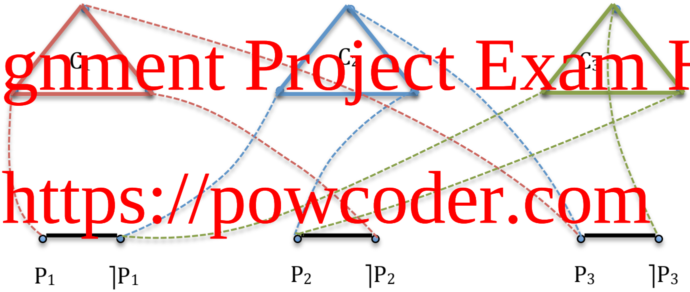
Reducing 3SAT to VC

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



Reducing 3SAT to VC

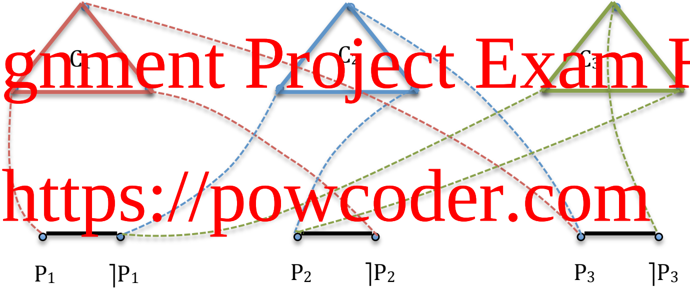
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



- Claim: an instance of 3SAT consisting of M clauses and containing in total N propositional variables has an assignment of variables which makes that instance true if and only if the corresponding graph has a Vertex Cover of size at most $2M + N$.

Reducing 3SAT to VC

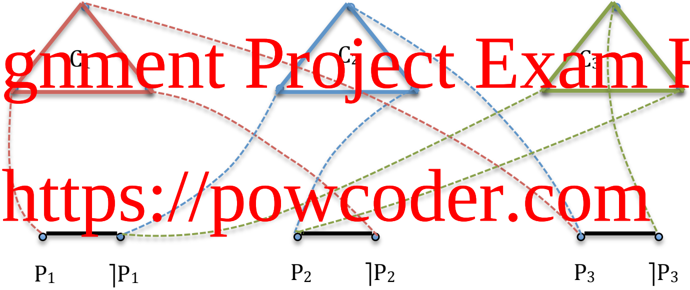
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



- Claim: an instance of 3SAT consisting of M clauses and containing in total N propositional variables has an assignment of variables which makes that instance true if and only if the corresponding graph has a Vertex Cover of size at most $2M + N$.
- Assume there is a vertex cover with at most $2M + N$ vertices chosen. Then

Reducing 3SAT to VC

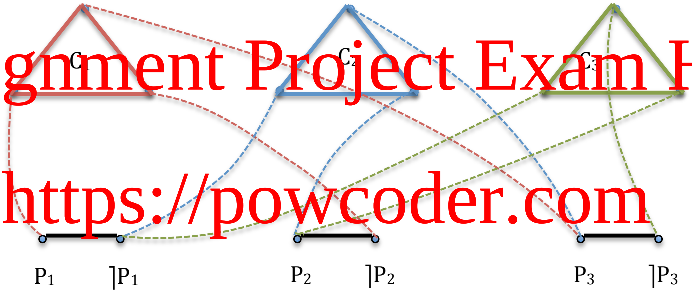
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



- Claim: an instance of 3SAT consisting of M clauses and containing in total N propositional variables has an assignment of variables which makes that instance true if and only if the corresponding graph has a Vertex Cover of size at most $2M + N$.
- 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

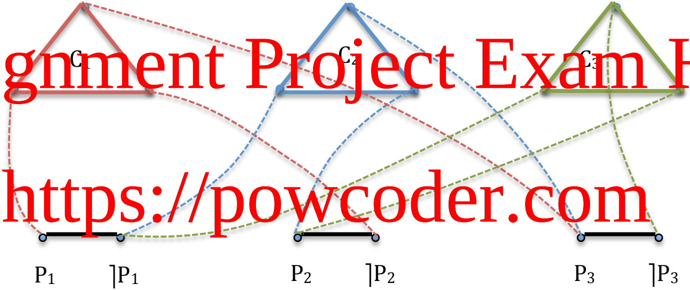
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



- Claim: an instance of 3SAT consisting of M clauses and containing in total N propositional variables has an assignment of variables which makes that instance true if and only if the corresponding graph has a Vertex Cover of size at most $2M + N$.
- 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

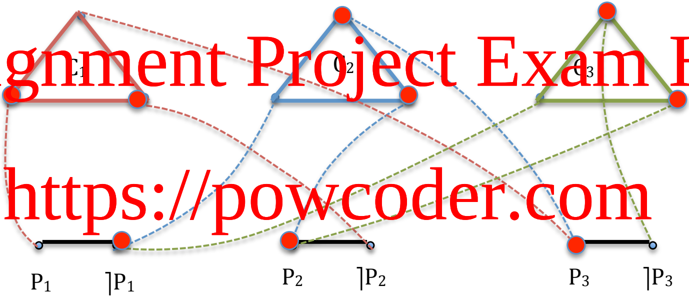
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



- Claim: an instance of 3SAT consisting of M clauses and containing in total N propositional variables has an assignment of variables which makes that instance true if and only if the corresponding graph has a Vertex Cover of size at most $2M + N$.
- 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

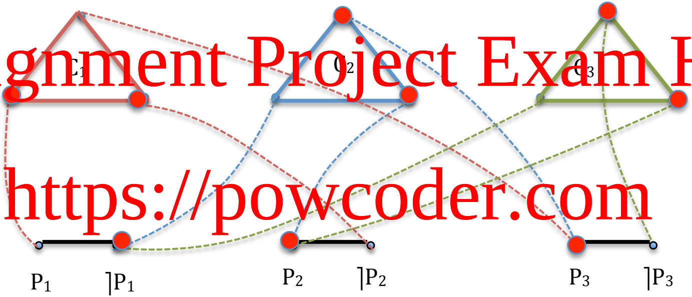
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



- Set each propositional letter P_i to true if P_i end of the segment corresponding to P_i is covered;

Reducing 3SAT to VC

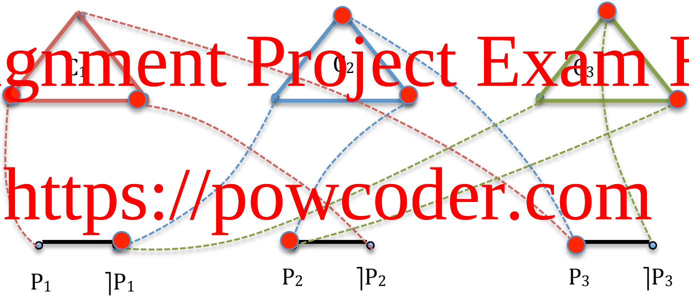
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



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

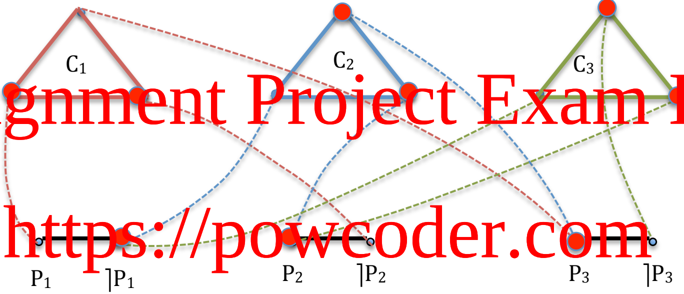
Reducing 3SAT to VC

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



- Set each propositional letter P_i to true if P_i end of the segment corresponding 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.

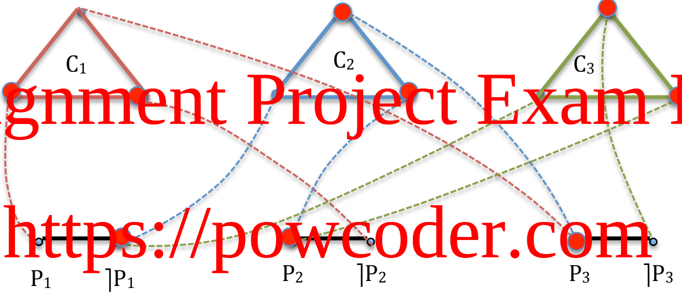
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



- Opposite, assume that the formula has an assignment of the variables which makes it true.

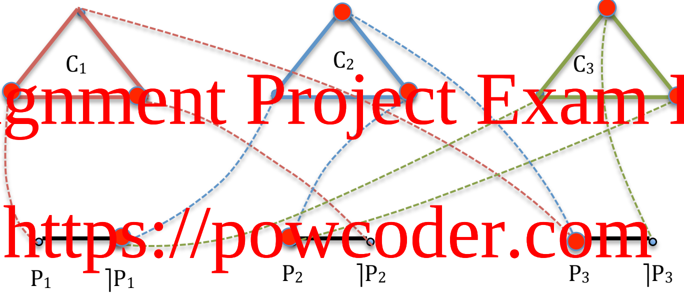
Reducing 3SAT to VC

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



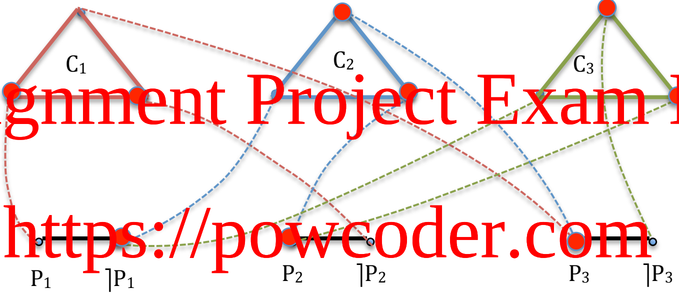
- Opposite, assume that the formula has an assignment of the variables which makes it true.
- For each segment, if it corresponds to a propositional letter P_i which such a satisfying evaluation sets to true cover its P_i end.

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



- Opposite, assume that the formula has an assignment of the variables which makes it true.
- For each segment, if it corresponds to a propositional letter P_i which such a satisfying 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.

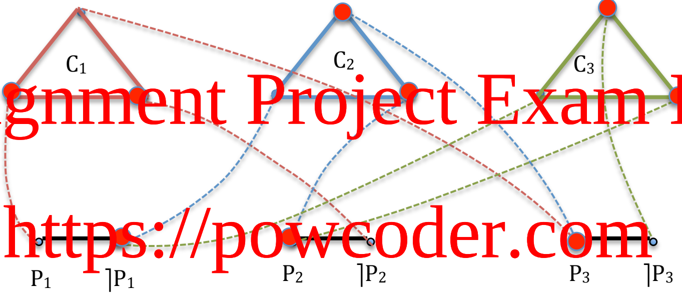
$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$



- Opposite, assume that the formula has an assignment of the variables which makes it true.
- For each segment, if it corresponds to a propositional letter P_i which such a satisfying 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

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



- Opposite, assume that the formula has an assignment of the variables which makes it true.
- For each segment, if it corresponds to a propositional letter P_i which such a satisfying 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.

- 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.
- Algorithm: (somewhat counter intuitive!)

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.
- Algorithm: (somewhat counter intuitive!)
  1. Pick an arbitrary edge and cover BOTH of its ends.

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.
- Algorithm: (somewhat counter intuitive!)
  1. Pick an arbitrary edge and cover BOTH of its ends.
  2. Remove all the edges whose one end is now covered. In this way you are left only with edges which have either both ends covered or no 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.
- Example: Vertex Cover has an approximation algorithm which always produces a covering set with at most twice the number of the smallest vertex cover.
- Algorithm: (somewhat counter intuitive!)
  1. Pick an arbitrary edge and cover BOTH of its ends.
  2. Remove all the edges whose one end is now covered. In this way you are left only with edges which have either both ends covered or no end covered.
  3. Repeat picking edges with both ends uncovered until no edges are left.

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.
- Algorithm: (somewhat counter intuitive!)
  1. Pick an arbitrary edge and cover BOTH of its ends.
  2. Remove all the edges whose one end is now covered. In this way you are left only with edges which have either both ends covered or no end covered.
  3. Repeat picking edges with both ends uncovered until no edges are left.
- Clearly, this produces a vertex cover because edges are removed only if one of their end is covered and we perform this procedure until no edges are left.

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.
- Algorithm: (somewhat counter intuitive!)
  1. Pick an arbitrary edge and cover BOTH of its ends.
  2. Remove all the edges whose one end is now covered. In this way you are left only with edges which have either both ends covered or no end covered.
  3. Repeat picking edges with both ends uncovered until no edges are left.
- Clearly, this produces a vertex cover because edges are removed only if one of their end is covered and we perform this procedure until no edges are left.
- 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.
- Algorithm: (somewhat counter intuitive!)
  1. Pick an arbitrary edge and cover BOTH of its ends.
  2. Remove all the edges whose one end is now covered. In this way you are left only with edges which have either both ends covered or no end covered.
  3. Repeat picking edges with both ends uncovered until no edges are left.

- Clearly, this produces a vertex cover because edges are removed only if one of their end is covered and we perform this procedure until no edges are left.

- 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.

- Example: Metric Traveling Salesman Problem (MTSP).

- 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

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

- Claim: MTSP has an approximation algorithm producing a tour of total length at most twice the length of the optimal, minimal length tour, denoted by *opt*.

- 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

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

- Claim: MTSP has an approximation algorithm producing a tour of total length at most twice the length of the optimal, minimal length tour, denoted by *opt*.

- Algorithm: Find a minimum spanning tree T of G. Since the optimal tour with one of its edges e removed represents a spanning tree, we have that the total weight of T satisfies $w(T) \leq opt - w(e) \leq opt$.

- 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

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

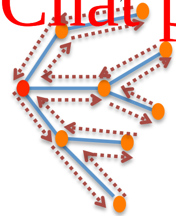
- Claim: MTSP has an approximation algorithm producing a tour of total length at most twice the length of the optimal, minimal length tour, denoted by *opt*.
- Algorithm: Find a minimum spanning tree T of G. Since the optimal tour with one of its edges e removed represents a spanning tree, we have that the total weight of T satisfies $w(T) \leq opt - w(e) \leq opt$.
- If we do depth first traverse of the tree, we will travel in total a distance of $2w(T) \leq 2opt$.

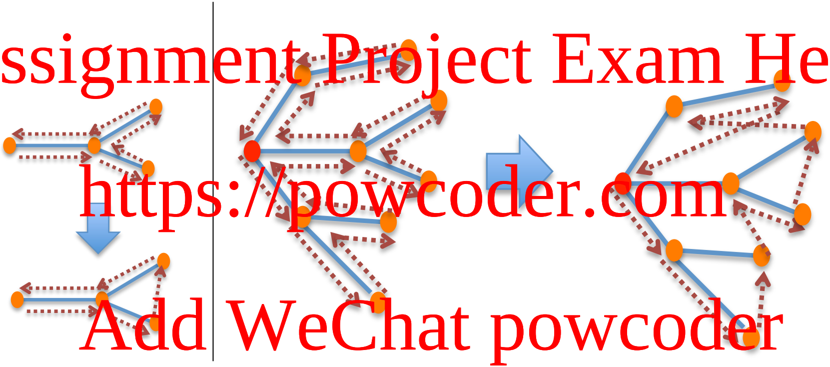
Dealing with NP hard optimisation problems

- 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

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

- Claim: MTSP has an approximation algorithm producing a tour of total length at most twice the length of the optimal, minimal length tour, denoted by opt.
- Algorithm: Find a minimum spanning tree T of G. Since the optimal tour with one of its edges e removed represents a spanning tree, we have that the total weight of T satisfies $w(T) \leq opt - w(e) \leq opt$.
- If we do depth first traverse of the tree, we will travel in total a distance of $2w(T) \leq 2opt$.





- 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 cover which is at most twice as large as the optimal cover of minimal possible 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 cover which is at most twice as large as the optimal cover of minimal possible size.

- On the other hand, the most general Traveling Salesman Problem does not allow any approximate solution at all: if $P \neq NP$, then for no $K > 0$ there can be a polynomial time algorithm which for every instance produces a tour which is at most K times longer than the optimal tour of minimal possible length, no 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 cover which is at most twice as large as the optimal cover of minimal possible size.
- On the other hand, the most general Traveling Salesman Problem does not allow any approximate solution at all: if $P \neq NP$, then for no $K > 0$ there can be a polynomial time algorithm which for every instance produces a tour which is at most K times longer than the optimal tour of minimal possible length, no matter how large K is chosen!
- To prove this, we show that if there existed $K > 0$ and a polynomial time 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.

Dealing with NP hard optimisation problems

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

- 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 $K \cdot opt$ where opt is the length of the optimal tour through G^*.

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 $K \cdot opt$ where opt is the length of the optimal tour through G^*.
- Clearly, if the original graph G has a Hamiltonian cycle then G^* has a tour consisting of edges already in G and of weights equal to 1, so such a tour has length of exactly 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 $K \cdot opt$ where opt is the length of the optimal tour through G^*.
- Clearly, if the original graph G has a Hamiltonian cycle then G^* has a tour consisting of edges already in G and of weights equal to 1, so such a tour has length of exactly n.
- Otherwise, if the original graph G does not have a Hamiltonian cycle, then the optimal tour through G^* must contain at least one added edge of length $K \cdot n$.

- 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 $K \cdot opt$ where opt is the length of the optimal tour through G^*.
- Clearly, if the original graph G has a Hamiltonian cycle then G^* has a tour consisting of edges already in G and of weights equal to 1, so such a tour has length of exactly n.

- Otherwise, if the original graph G does not have a Hamiltonian cycle, then the optimal tour through G^* must contain at least one added edge of length $K \cdot n$.
- In this case the optimal tour through G^* has length of at least $(K \cdot n) + (n-1) \cdot 1 > K \cdot n$.

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 $K \cdot opt$ where opt is the length of the optimal tour through G^*.
- Clearly if the original graph G has a Hamiltonian cycle then G^* has a tour consisting of edges already in G and of weights equal to 1, so such a tour has length of exactly n.
- Otherwise, if the original graph G does not have a Hamiltonian cycle, then the optimal tour through G^* must contain at least one added edge of length $K \cdot n$.
- In this case the optimal tour through G^* has length of at least $(K \cdot n) + (n-1) \cdot 1 > K \cdot n$.
- 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 $K \cdot opt$ where opt is the length of the optimal tour through G^*.

- Clearly, if the original graph G has a Hamiltonian cycle then G^* has a tour consisting of edges already in G and of weights equal to 1, so such a tour has length of exactly n.
- Otherwise, if the original graph G does not have a Hamiltonian cycle, then the optimal tour through G^* must contain at least one added edge of length $K \cdot n$.

- In this case the optimal tour through G^* has length of at least $(K \cdot n) + (n - 1) \cdot 1 > K \cdot n$.
- 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.